NATIONAL

Order Number IPC-16A/927
Publication Number 4200127X
Preliminary

The PACE Microprocessor

A Logic Designer's Guide to Program Equivalents of TTL Functions

MARCH 1976

National Semiconductor Corporation
2900 Semiconductor Drive
Santa Clara, California 95051

National does not assume any responsibility for use of any circuitry described; no circuit patent licenses are implied; and National reserves the right, at any time without notice, to change said circuitry.

A50M36/Printed in U.S.A.

PREFACE

This handbook is intended for the TTL system designer; it shows him how standard TTL/MSI logic functions are implemented in software for the PACE microprocessor. This handbook in fact describes two classes of hardware simulation by PACE.

The first class describes the simulation of standard, single-package TTL functions (e.g., a DM74154 4-line to 16-line decoder/demultiplexer) by software routines (although about half of this class are examples of multiple-package extensions of standard 4-bit functions to 16 bits), while the second class describes the simulation in software of multiple, "non-standard," subsystem functions (e.g., a tachometer comprised of four DM7413 binary counters, four DM7485 comparators, and four DM7123 multiplexers.

With one exception, the second class of simulations – the subsystems – are presented as a single entity. That is, the routines for the various building blocks of the subsystem are not presented individually; instead, a single software solution is presented as a cohesive whole in much the same way that a designer (one used to thinking in terms of software) would approach the problem.

To bridge the gap between the single-package simulations and the subsystem simulations – that is, to show the linking of the subsystem's building blocks – one subsystem simulation (the digital servo) is presented in both a step-by-step manner (to show its building-block components, their subroutines, and how these subroutines are meshed to form the complete subsystem function), and as a final, single routine that is a somewhat more elegant blend of its component parts.

All of the simulations have been desk-checked and assembled; in fact, this handbook reproduces the actual "no error" assembled program print-outs. This is by no means a guarantee that any given simulation will run immediately on any given PACE system; this no-run phenomenon, common to all software-controlled systems, is explained on page 1-5.

All simulations in this handbook conform to several ground rules. For the standard TTL function simulations, it is assumed that:

1. Input conditions are set into, and outputs formed in, one of PACE's four accumulators (AC0, AC1, AC2, or AC3);
2. The result of the operations – the output – is left in an accumulator (i.e., transfers to and from memory or peripherals are not shown); and,
3. The interrupt, flag, and jump-condition capabilities of the PACE microprocessor are not used.

   For subsystem simulations, the first two rules (nos. 1 and 2) remain in effect, but rule no. 3 is voided: interrupts, flags, and jump conditions are exploited.

Note that for the TTL counter simulations we bend (slightly) rule no. 3 so that the carry flag (status register bit 7) is set to indicate the finish of the count sequence; the simulations, in fact, are written in a way that ensures the carry flag will be reset by every subroutine call that does not result in completion of the count sequence.

In practice, however, instructions associated with a carry-flag reset may be unnecessary, as such resets are needed only when the carry flag either is tested following every return to the main program or is automatically included as an input by a DECA or SUBB instruction following the subroutine.

Where applicable, each DECA or SUBB instruction within *any* subroutine is preceded by a reset of the carry flag (PFLG 15 instruction); again, this procedure may not be needed in practice if you know that the carry flag is in the reset state when the subroutine is called by the main program.

The programs in this book have been assembled in relocatable mode, rather than in absolute mode. In relocatable mode, the starting address of the program is defined when the binary object code (of the assembled program) is loaded into memory by the loader program; in absolute mode the starting address is defined when the program is assembled.

If an absolute program had been loaded starting at, say, location X'100, but the programmer now wants to load the program starting at, say, location X'200, he or she must reassemble the program with the new starting address. A relocatable program, on the other hand, may be loaded starting at location X'100, X'200, or any other location.

The programmer normally would use an absolute-sector (.ASECT) directive in the program to indicate absolute mode, or a base-page-sector (.BSECT) or top-page-sector (.TSECT) directive to indicate relocatable mode. But since the PACE assembler initializes in the top-page-sector relocatable mode, a directive is not required.

TABLE of CONTENTS

CHAPTER 1. A BRIEF INTRODUCTION TO MICROPROCESSING

CHAPTER 2. THE PACE INSTRUCTION SET

CHAPTER 3. MICROPROCESSOR INS AND OUTS

CHAPTER 4. THE SIMULATIONS

Standard Functions

Subsystems

APPENDICES

Chapter 1
A BRIEF INTRODUCTION TO MICROPROCESSING

CHAPTER 1–A BRIEF INTRODUCTION TO MICROPROCESSING

Today, a computer connotes a machine that, once it is set up for a specific problem, performs a computation automatically and without human intervention. The present use of the term "computer" has a second connotation—it usually refers to an electronic machine, although mechanical and electromechanical computers do exist. Two important factors dictate the intimate association between computers and electronics: no known principle other than electronics allows a machine to attain the speeds now commonplace in both large- and small-scale computers; and, no other principle permits comparable design convenience. In particular, digital computers use numbers that are represented by the presence or absence of a voltage level or pulse on a given signal line. A single pulse defines one "bit" (short for binary digit, a base-2 number); a group of pulses considered as a unit is called a "word", where a word may represent a computational quantity or a machine directive.

For purposes of illustration, we shall compare two systems for solving simple mathematical expressions, both of which are comprised of the classical elements of a computer: an input/output device, a memory, a control section, and an arithmetic and logic unit or ALU (the computational element). The control section, together with the ALU, is considered to be the central processing unit (CPU). (See *Figure 1*)

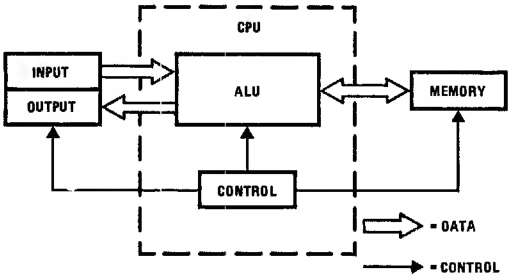


FIGURE 1. Basic Elements of a Digital Computer

THE MAN-CALCULATOR

The first system *(Figure 2)* is comprised of a man and a calculator. The man's fingers represent the input, his eyes coupled with the calculator's output represent the system output, the calculator electronics function as the ALU, and his brain serves as the memory as well as the control section. Here is the sequence of events that occurs when our man-calculator solves the problem 6 + 2 = ?

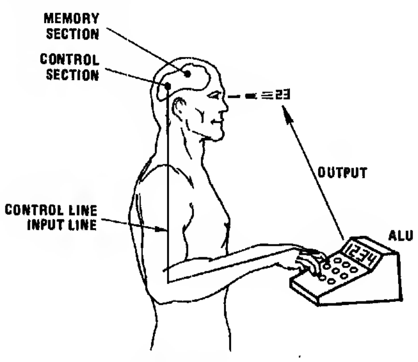


FIGURE 2. Man + Calculator = Computer

1. Brain accesses first number to be added, a "6";
2. Brain orders hand to depress "6" key;
3. Brain identifies addition operation;
4. Brain orders hand to depress "+" key;
5. Brain accesses second number to be added, a "2";
6. Brain determines that all necessary information has been provided and signals the ALU to complete computation by ordering hand to depress "=" key;
7. ALU (calculator) makes computation;
8. ALU displays result on readout;
9. Eyes signal brain, brain recognizes this number as the result of the specific calculation;
10. Brain stores result, "8", in a location that it appropriately identifies to itself to facilitate later recall.

THE CLASSICAL COMPUTER

We shall now develop a classical computer and illustrate how it might be used to solve the same problem. To begin, note that the memory (*Figure 3*) is composed of storage space for a large number of words; each storage space is identified by a unique "address". The word stored at a given address may be either computational data or a machine directive (such as add, read from memory, etc.). Two temporary storage registers, each capable of containing one word, complete the memory. These registers are designated as "memory address register" (MAR) and "memory data register" (MDR). The MAR contains the binary representation of the address at which information is to be read out of memory or written (stored) into memory, while the MDR contains the data being exchanged with memory.

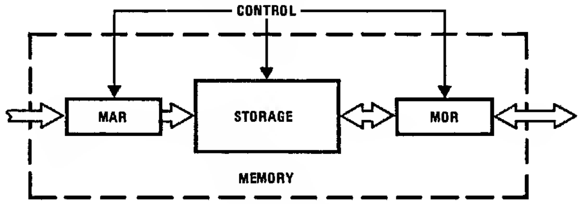


FIGURE 3. Elements of a Memory

Turning to the ALU, *Figure 4* shows that this portion of a computer, in its simplest form, comprises an "adder" that adds (or performs similar logical operations upon) two inputs A and B and produces an output at C, and an "accumulator", which maintains intermediate results of a computation or numbers for a pending computation.

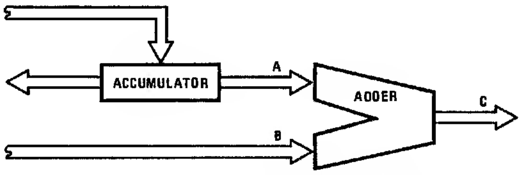


FIGURE 4. Arithmetic and Logic Unit

The remainder of the CPU, the control portion, is implemented using an "instruction register" (IR), a "control decoder and sequencer", and a program counter (PC). These are shown in *Figure 5.* A machine directive (instruction) is transferred into the IR and is subsequently interpreted by the decoder/sequencer, which issues the appropriate control pulses to the other computer elements. The PC contains, at any given time, the address in memory of the next machine directive or instruction. This counter is normally incremented by a count of one immediately following the reading of a new instruction. The PC contents may be replaced by the contents of a specified memory location if the last instruction was of the "jump" class. This causes the next instruction to be read from a program-specified location, instead of from the next sequential location as is the general rule.

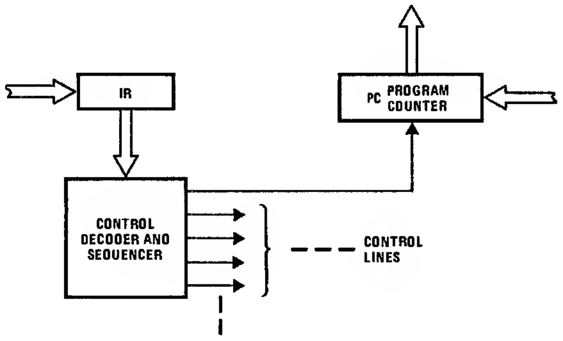


FIGURE 5. Computer Control

Finally, a means of input/output (I/O) is provided by an "I/O Register", through which data is exchanged with external (peripheral) devices. *(Figure 6.)*

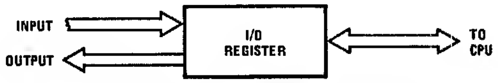


FIGURE 6. I/O Register Interface

We have now collected all the basic elements of a computer; all that remains to do is to interconnect them into a functioning, automatic processor. *Figure 7* shows such an interconnection, and represents a complete computer.

The analysis continues with the execution of the same problem used to illustrate the man-calculator, but somewhat rephrased:

> "Read-in a number from the I/O. Store it in memory location 50. Read-in another number from the I/O. Add the two numbers together. Store the result in memory location 60, and halt."

A "program" has been written to execute this task, and is stored in consecutive memory locations beginning at 100. This program, written in an artificial symbolic language, is shown in Table 1.

TABLE 1. Sample Program

Memory Location	Instruction (Contents)
100	Input to accumulator
101	Store accumulator at 50
102	Input to accumulator
103	Add accum, Loc. 50 Place result in accumulator
104	Store accumulator at 60
105	Halt

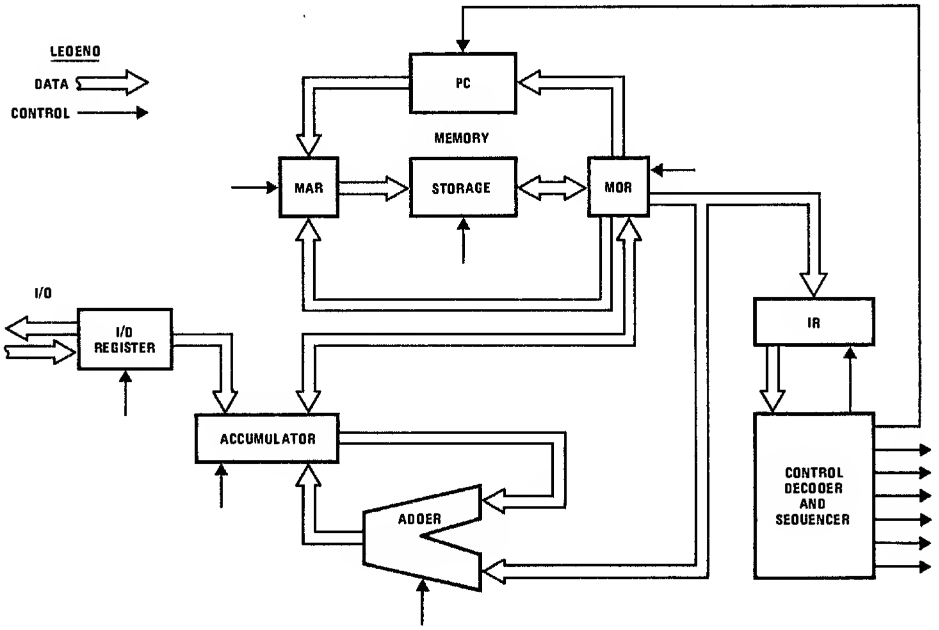


FIGURE 7. Simplified CPU and Memory

Computer States

All computers spend about equal periods of time in one of two distinct states: "fetch", or "execute". In the fetch state, the computer reads from memory the next sequential instruction and places it in the instruction register (IR). In the execute state, that instruction is carried out as a series of transfers from one register to another and as various ALU operations. Table 2 examines the program shown in Table 1, as it is actually executed, by specifying the contents of each register at each machine cycle (time interval) and assuming the computer is now ready to fetch the first instruction in our program.

All computers (processors, CPU's, etc.) operate in a similar manner, regardless of their size or intended purpose, although many variations are possible within the basic architectural framework. Common variations include, for example, highly-sophisticated I/O structures (some of which have direct and/or autonomous communications with memory), multiple accumulators for programming flexibility, index registers that allow a memory address to be modified by a computed value, multi-level interrupt capability, and on and on.

MICROPROGRAMMING

One of the most exciting architectural concepts to gain popularity in the past few years is that of microprogrammed control. A microprogrammed computer differs from the classical example in its control-unit implementation. The classical machine has for its control unit an assemblage of logic elements (gates, counters, flip-flops, etc.) interconnected to realize certain combinatorial and sequential Boolean equations. On the other hand, a microprogrammed machine uses the concept of a "computer within a computer." That is, the control unit has all the functional elements that comprise a classical computer, including read-only memory (ROM).

The "inner computer", which (generally) is not apparent to the user, executes the user's program instructions by executing a series of its own microinstructions, thereby controlling data transfers and all functions from computed results. And this means that changing the stored microprogram that generates the control signals alters the entire complexion of the computer. By altering a few words stored in the ROM, the computer behaves in an entirely new fashion — it can execute a completely different set of instructions, simulate other computers, tailor itself to a specified application. It is this capability for "custom-tailoring" that allows a microprogrammed machine to be optimized for a given usage. By so extracting the utmost measure of efficiency, a microprogram-controlled machine is less costly and easier to adapt to any given situation, no matter how diverse or demanding.

Software and the Microprocessor

It is possible to program a device that isn't a computer at all. An operational amplifier, for example, is a circuit that is basically a multiplier. Something is put in, something comes out; the op amp performs a linear function. But this building block can do something other than multiplication: a capacitor, for example, connected from the op amp's output to its input, creates a "programmed-by-wire" integrator.

As it is with the op amp, so it is with the microprocessor. A microprocessor is a super circuit—a black box with a transfer function that changes in accordance with a set of commands called a program. Inside the black box (i.e., on the chip) is a collection of building-block logic—an assemblage of many logic elements. You can in fact replace the microprocessor in any system with sets of random logic on PC boards, but you would have to change the logic boards on each clock pulse!

Thus, if you know what a flip-flop does you know what it does inside or outside a microprocessor; an AND gate ANDs whether it's inside a microprocessor or on a lab bench. But in a microprocessor literally thousands of such logic elements are squeezed onto one or two chips. And this creates a problem: too much information, too few pins.

TABLE 2. Register Content

NOTES	PC	ACCUM.	MAR	MDR	I/O REG.	IR	MEMORY (R=READ) (W=WRITE)	STATE
	100	?	?	?	?	?	?	?
Start	100	?	100	(100)	?	(100)	R	Fetch
Input	100	6	100	(100)	6	(100)		Execute
	101	6	101	(101)	?	(101)	R	Fetch
Store	101	6	50	6	?	(101)	W	Execute
	102	6	102	(102)	?	(102)	R	Fetch
Input	102	2	102	(102)	2	(102)		Execute
	103	2	103	(103)	?	(103)	R	Fetch
	103	2	50	6	?	(103)	R	Fetch
Add	103	8	50	6	?	(103)		Execute
	104	8	104	(104)	?	(104)	R	Fetch
Store	104	8	60	8	?	(104)	W	Execute
	105	8	105	(105)	?	(105)	R	Fetch
Halt	105	8	105	(105)	?	(105)		Execute

To overcome the pin problem, microprocessor manufacturers strap every logic element to every other logic element through a set of buses that allows mutual, element-to-element communications. Bus connections are made through a series of electronic switches; opening and closing the switches transfers the data through the microprocessor's maze to produce a control function. And it is software that sets the switches. System software is a set of tools, supplied by the microprocessor manufacturer, that allows you to construct application programs—programs that let the microprocessor do something.

To appreciate what software does for you, consider an elementary operation such as addition. Get A, get B, add them together and come out with C. Easy? In decimal notation, yes. But this trivial problem is not quite as simple when one speaks in binary. Dealing with long binary numbers is complex and difficult because one's and zero's aren't a natural language for Homo Sapiens. We have problems trying to figure out what's going on when we look at raw binary; writing it is even more troublesome.

Can you imagine looking down 14 sheets of printout, each with 65 lines of binary gibberish, attempting to determine what you did wrong? Yet this is ultimately how you program a job on a microprocessor. You have to write the story of how the processor is to wire itself from microsecond to microsecond. So all system software, the whole range of it that every manufacturer offers, is aimed at only one thing: to get you from the stated idea to the working program as painlessly and as rapidly as possible.

The Software Process

In the construction of application software, you first evolve a flowchart (*Figure 8A*) that describes the functions to be performed and their order. (At this stage your thought processes and activities resemble those of the random-logic designer.) Once the chart is laid out, you start to code the program in either a high-level or a mnemonic-shorthand language that both you and your system understand. Here you encounter your first piece of software, the Text or Source Editor (*Figure 8B*).

Most microprocessor users write on continuous media (paper tape or cassettes), which do not allow you to get in and pull out one piece. Thus, corrections on a continuous source involve making a wholly new source—a constant problem and an awfully wasteful task. But there is a utility program called a Source Editor that lets you do the entire job with a Teletype® and a microprocessor Development System. If you make an error, just tell the Editor what changes to make and it's done! The Editor helps you massage the source code until it looks like it's going to work. Then, with the corrected (?) program in the Editor's memory cells, you push a button and a paper tape (or whatever) is put into your hands.

The "whatever" that has just been put into your hands has one minor, relatively insignificant, but fatal error—the microprocessor cannot understand a single bit or byte of it. But do not despair: an electronic Translator (*Figure 8C*) converts the continuous, source-mnemonic shorthand into something the microprocessor can understand.

The Translator (*Figure 9A*) takes the source tape and gives back three outputs:

- The Program Listing—a copy of both the source and binary object codes;
- The Error Listing—a roster of all grammatical, label, and syntax errors; and,
- The Binary Object Code—a paper tape (or whatever) with the machine-readable binary translation of the program.

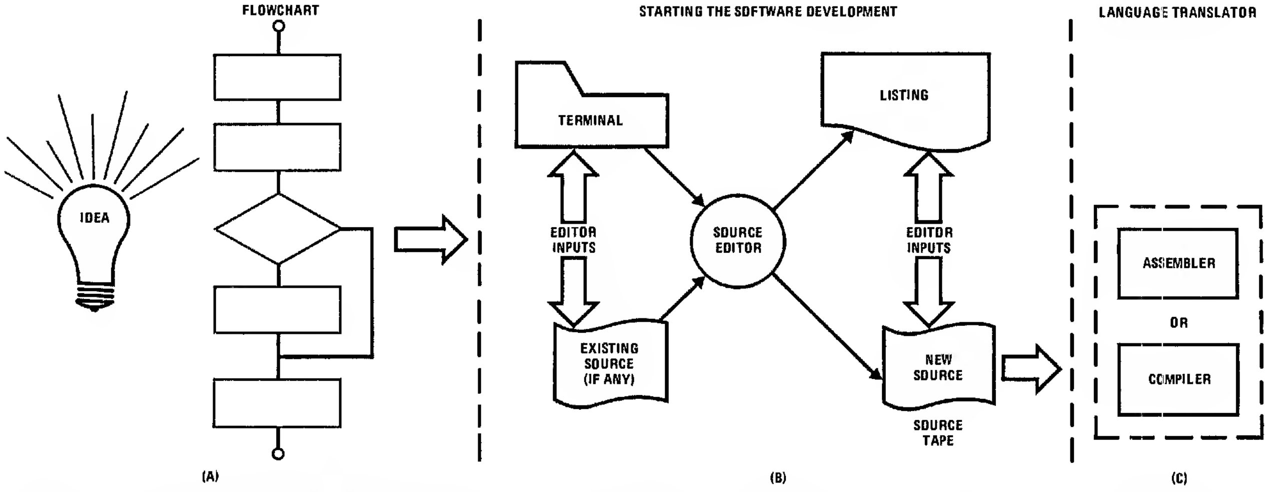


FIGURE 8. The programmer's ideas, expressed in a flowchert, ere written out in mnemonic form to serve es Editor inputs. New inputs plus sections of existing programs ere combined to form a new Source; this Source is the input to the language Translator.

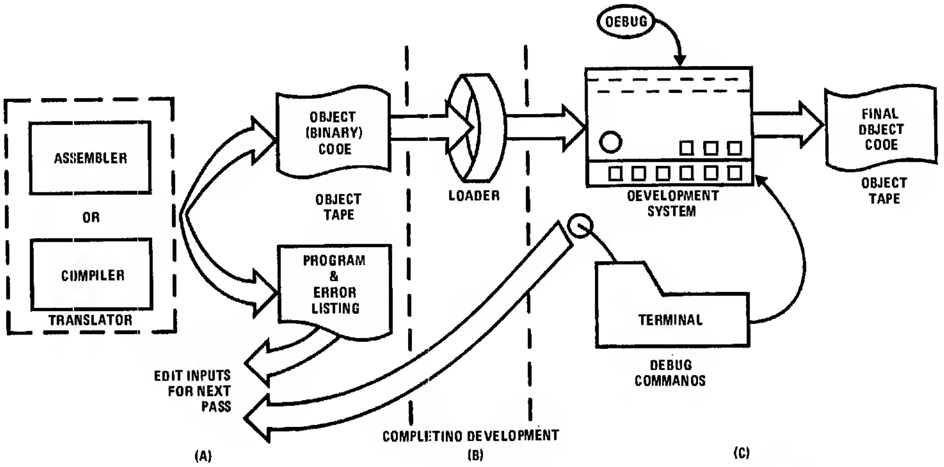


FIGURE 9. **Trensletor outputs include: an Error Listing (to serve as Editor inputs on the next pess); a Program Listing; and a tapa of the trenslated program |tha Object Tepe). The Objact Tape is deposited by tha Loeder into Read/Write Mamory inside the Development System. Here the new code is run by the DEBUG progrem eccording to commands input by you. Tha code can be modified via terminal inputs until it runs properly; workIng coda is then dumpad from memory. Nota thet elthough a workeble object tape may exist at this time, your job is not cornplate until you edit end ratranslete your Source to produce code identical to tha working code.**

But there are two types of Translators—the Assembler and its exotic cousin, the Compiler—and there may be some argument as to which translation device is the more useful: Should you use an Assembler or a Compiler to translate the mnemonic source? The difference is in the mnemonics.

If you happen to have run programs on minicomputers, then you've been exposed to the so-called "assembly language" mnemonics: LD means load; JMP means jump; ST means store; etc. It's the shortest language (outside of raw binary) used to talk with the processor. Programming with this shorthand is a bit tricky but an assembler-type Translator gives you a better feel for the machine and you can usually pare down the number of statements necessary to get the message across; and this saves time and money.

On the other hand, a compiler-type Translator lets you write in a high-level language that looks like English (Fortran, etc.). Its statements can easily be read by someone with no training at all. The Compiler translates these statements into a series of machine commands that carry out the desired function with the advantages of faster programming and a self-documenting program that you can read directly. But you often pay for this ease of use: since the Compiler deals with more general statements, it often translates in an inefficient way using more machine commands than really necessary at that level. Extra statements consume memory and result in slower program execution.

So, in retrospect, Compilers cut programming time and costs, but raise system costs. Assemblers do just the opposite. Which should you use? Compilers are most useful to those of you who constantly re-program your systems and make few versions of each program. Assembler users, on the other hand, will be those of you who will program the system once, then reproduce it a thousand or more times; programming costs are amortized over the production run and in memory savings.

At this point in the writing of a program many of you will wish that you could forget the whole thing, for there are programs with one hundred code lines that come out of the Translator with four hundred errors! But forge onward. Make another pass through the Source Editor (and another, and another. . .), to correct the errors that the Translator has spotted. Eventually, you will get your reward, the sweetest line ever printed on a computer listing: "ASSEMBLY COMPLETE – NO ERRORS." Actually, that statement simply means that the Assembler didn't find any errors. And you soon find out that this has almost nothing to do with whether or not the program will run on a machine. The reason is that the Assembler, although it helps you weed out logic errors from the program that you wrote, cannot tell you whether or not that program does exactly what you think it's going to do. In other words, there can be (and very probably will be) logic differences between your vision (of what's needed to perform a function) and that of the machine. Such an error may be one as simple as your forgetting to set a flag at some point; unimportant, perhaps, to your charting of a problem's solution, but all-important to the machine for without that bit of information your program cannot run. But other utility programs (such as DEBUG) are available to help you solve such problems.

Now that the Translator has provided you a binary tape with your program on it, you must somehow get the program into the machine's memory along with whatever other software routines your program needs for operation. The Loader (*Figure 9B*) does this for you; it reads your tape into a microprocessor Development System (*Figure 9C*), allocates memory space to the program, and stores the code in the appropriate location. Typically, several sections of memory are needed for different functions (executable code, interrupt calls, subroutine linkages, etc.), and it is up to the Loader to see that each part of the program is put into the right place. Loaders are available to load from Teletypes, paper-tape readers, and, sometimes, high-speed bulk storage devices.

Once the program is loaded, you cross your fingers and hit the RUN switch. As we've already said, very probably nothing will happen.

Now, if you are using random logic and find it doesn't work, you unplug it, repair any damaged hardware, and then try to determine what's wrong. With an oscilloscope on the gates and clocks, you try to see what's happening. But in the microprocessor only one set of logic exists, re-wiring itself at the speed of light. If you don't have any idea what's going on, the oscilloscope can't help you. What you need is a different type of fault-finding tool. The tool is a program, called DEBUG, that lets you use a Teletype as a scope to help you find out what's happening. DEBUG is loaded into a Development System first, then your program is entered. You peck away at the TTY and say, "DEBUG, run my program from here to there, stop it, and tell me what is in memory." The TTY rattles and you've got the answer on a printout. "Show me what is in these accumulators." DEBUG does! "Show me this, show me that." Done, done. As your program is stepped through, you'll encounter parts that don't work. These snags are cajoled and fondled individually until the whole thing runs—perfectly—and you have a working object code that represents your algorithm in ones and zeros.

There is an alternative to the microprocessor debug section of a Development System. It is called a Simulator, and it typically runs on a large computer and includes both debug and simulation. To use it, load the binary code into the computer, call the Simulator, and then direct it to exercise the code to find the defects. However, this approach can only take you part of the way; it will not isolate timing problems that have to do with the outside world.

When the Simulator wants an input, it stops and asks for one. You sit there and peck away at the typewriter, which is fine if you want to test things that are slow. But if you wish to test a program that operates, say, a 100-kHz I/O converter, you won't be able to keep up with it. So the Simulator can only take you so far. Ultimately you have to return to the hardware prototype approach, and this is why the microprocessor manufacturers have felt it necessary to produce sophisticated hardware prototyping tools.

We at National believe a Simulator really doesn't help. We encourage users to take the Development System itself, put in the actual interfaces to be used, and use DEBUG to massage the program in real-time and watch what it does.

Chapter 2
THE PACE INSTRUCTION SET

CHAPTER 2 – THE PACE INSTRUCTION SET

This chapter contains detailed descriptions of the instructions provided by the PACE microprocessor. The PACE microprocessor provides a general purpose mix of 45 instructions, which are divided into eight format groups as follows:

- Branch instructions
- Skip instructions
- Memory data-transfer instructions
- Memory data-operate instructions
- Register data-transfer instructions
- Register data-operate instructions
- Shift and rotate instructions
- Miscellaneous instructions

Many of the 45 instructions comprising the eight format groups could be generally classified as falling into one of three instruction classes:

- Memory-reference instructions
- Register instructions
- Data-transfer instructions

The memory-reference instructions use a flexible memory addressing scheme that provides three floating memory pages of 256 words each and one fixed memory page of 256 words. The register instructions provide a convenient means of data manipulation without accessing memory. The data-transfer instructions provide a convenient means of moving data among the functional blocks of the PACE microprocessor system.

In the PACE microprocessor, data is represented in the twos-complement number system, in which the negative of a number is formed by complementing each bit and, then, adding one to the complemented value of the number. The most-significant bit position indicates the sign of the number, 0 for positive and 1 for negative. With a single 16-bit value, the greatest positive number is X'7FFFF or $(32767)_{10}$, and the most negative number is X'8000 or $(32768)_{10}$. When the 8-bit data length is selected, the largest positive number is X'7F or $(127)_{10}$, and the most negative number is X'80 or $(128)_{10}$.

Both direct and indirect memory addressing instructions are included in the PACE instruction set. Direct memory addressing has three available modes: base-page; Program-Counter (PC) relative; and, indexed. The addressing mode is specified by the xr field of the instruction as illustrated in *Figure 10.*

15 10	9 8	7 0
OPERATION (op code)	Index (xr)	DISPLACEMENT (disp)

NS10278

FIGURE 10. Memory-Reference Instruction Format

When the xr field is 00, base-page (page zero) addressing is specified. Two types of base-page addressing are available. The type of base-page addressing selected is determined by the state of the Base-Page Select Signal (BPS) input. When BPS is low (0), the 16-bit memory address is formed by setting bits 8 through 15 to zero and using the 8-bit displacement (disp) field for bits 0 through 7. Thus, the first 256 words of memory (locations 0 through 255) can be addressed. When BPS is high (1), the 16-bit memory address is formed by setting bits 8 through 15 equal to bit 7 of the disp field and using disp for bits 0 through 7. Thus, the first 128 words (0 through 127) and the last 128 words (X'FF80 through X'FFFF) of memory can be addressed. The latter technique is useful for splitting the base page between read/write and read-only memories or between memory and peripheral devices. Consequently, base-page addressing provides a convenient means of accessing data or peripherals.

When the xr field is 01, addressing relative to the PC is specified. During the PC-relative addressing mode, the memory address is formed by adding the contents of PC to the value of the disp field, which is interpreted as a signed number. The 8-bit disp field is interpreted as a 16-bit value with the bit 7 value used for bits 8 through 15, thereby permitting representation of numbers from –128 through 127.

When the memory address is formed, the PC already is incremented and contains an address value that is one greater than the location of the current instruction. Thus, memory addresses that can be referenced range from 127 locations below through 128 locations above the address of the current instruction.

The indexed (or accumulator-relative) mode of addressing permits any memory location within the 65,536 word-address-space to be referenced. The disp field, as in PC-relative addressing, is interpreted as a signed value ranging from –128 through 127. The memory address is formed by adding disp to the contents of either Accumulator AC2 (when xr = 10) or Accumulator AC3 (when xr = 11). Table 3 presents a summary of the direct addressing modes.

TABLE 3. Summery of Direct Addressing Modes

xr FIELD	ADDRESSING MODE	EFFECTIVE ADDRESS
00	Base-Page	EA = disp
01	Program-Counter-Relative	EA = disp + (PC)
10	AC2-Relative (indexed)	EA = disp + (AC2)
11	AC3-Relative (indexed)	EA = disp + (AC3)

Note 1: For base-page addressing, disp is positive and in range of 000 to 255 when BPS is low (0); or disp is signed number in range of –128 to +127 when BPS is high (1).

Note 2: PC contains value one greater than address of current instruction.

Note 3: For relative addressing, display range is –128 to +127.

Indirect addressing consists of first establishing an address in the same manner as direct addressing (by either the base-page, PC-relative, or indexed mode). The contents of the memory location at the selected address then are used as the operand address.

NOTE: As explained in Chapter 2 of the PACE Users Manual, the memory addressing modes also are used for peripheral I/O operations. Address space must be divided between memory and I/O devices. Chapter 10 of that manual discusses addressing relevant to assembly language programming, and Chapter 7 discusses the address assignments used in the PACE Microprocessor Development System.

A summary of the 45 PACE instructions is provided in Table 4, which shows the instruction mnemonic, meaning, a symbolic representation of the instruction, the assembler format, and the instruction format. Table 5 defines the notation and symbols used in Table 4 and the remainder of this chapter. The notations are presented in alphabetical order and, then, the symbols are listed. Upper-case mnemonics refer to fields in the instruction word. Lower-case mnemonics refer to the numerical value of the corresponding fields. In cases where both upper- and lower-case mnemonics are composed of the same letters, only the lower-case mnemonic is given. The use of lower-case notation designates variables.

TABLE 4. PACE Instruction Summary

Mnemonic	Meaning	Operation	Assembler Format		Instruction Format			
1. **Branch Instructions**								
BOC	Branch On Condition	(PC) ← (PC) + disp if cc true	BOC	cc,disp	0 1 0 0	cc	disp	
JMP	Jump	(PC) ← EA	JMP	disp (xr)	0 0 0 1 1 0	xr	disp	
JMP@	Jump Indirect	(PC) ← (EA)	JMP	@disp (xr)	1 0 0 1 1 0			
JSR	Jump To Subroutine	(STK) ← (PC), (PC) ← EA	JSR	disp (xr)	0 0 0 1 0 1			
JSR@	Jump To Subroutine Indirect	(STK) ← (PC), (PC) ← (EA)	JSR	@disp (xr)	1 0 0 1 0 1			
RTS	Return from Subroutine	(PC) ← (STK) + disp	RTS	disp	1 0 0 0 0 0	0 0	disp	
RTI	Return from Interrupt	(PC) ← (STK) + disp, IEN = 1	RTI	disp	0 1 1 1 1 1			
2. **Skip Instructions**								
SKNE	Skip if Not Equal	If (ACr) ≠ (EA), (PC) ← (PC) + 1	SKNE	r,disp (xr)	1 1 1 1	r	xr	disp
SKG	Skip if Greater	If (AC0) > (EA), (PC) ← (PC) + 1	SKG	0,disp (xr)	1 0 0 1 1 1			
SKAZ	Skip if And is Zero	If [(AC0) ∧ (EA)] = 0, (PC) ← (PC) + 1	SKAZ	0,disp (xr)	1 0 1 1 1 0			
ISZ	Increment and Skip if Zero	(EA) ← (EA) + 1, if (EA) = 0, (PC) ← (PC) + 1	ISZ	disp (xr)	1 0 0 0 1 1			
DSZ	Decrement and Skip if Zero	(EA) ← (EA) – 1, if (EA) = 0, (PC) ← (PC) + 1	DSZ	disp (xr)	1 0 1 0 1 1			
AISZ	Add Immediate, Skip if Zero	(ACr) ← (ACr) + disp, if (ACr) = 0, (PC) ← (PC) + 1	AISZ	r,disp	0 1 1 1 1 0	r		
3. **Memory Data Transfer Instructions**								
LD	Load	(ACr) ← (EA)	LD	r,disp (xr)	1 1 0 0	r	xr	disp
LD@	Load Indirect	(AC0) ← ((EA))	LD	0,@disp (xr)	1 0 1 0 0 0			
ST	Store	(EA) ← (ACr)	ST	r,disp (xr)	1 1 0 1	r		
ST@	Store Indirect	((EA)) ← (AC0)	ST	0,@disp (xr)	1 0 1 1 0 0			
LSEX	Load With Sign Extended	(AC0) ← (EA) bit 7 extended	LSEX	0,disp (xr)	1 0 1 1 1 1			
4. **Memory Data Operate Instructions**								
AND	And	(AC0) ← (AC0) ∧ (EA)	AND	0,disp (xr)	1 0 1 0 1 0	xr	disp	
OR	Or	(AC0) ← (AC0) ∨ (EA)	OR	0,disp (xr)	1 0 1 0 0 1			
ADD	Add	(ACr) ← (ACr) + (EA), OV, CY	ADD	r,disp (xr)	1 1 1 0	r		
SUBB	Subtract with Borrow	(AC0) ← (AC0) + ~ (EA) + (CY), OV, CY	SUBB	0,disp (xr)	1 0 0 1 0 0			
DECA	Decimal Add	(AC0) ← (AC0) $+_{10}$ (EA) $+_{10}$ (CY), OV, CY	DECA	0,disp (xr)	1 0 0 0 1 0			
5. **Register Data Transfer Instructions**								
LI	Load Immediate	(ACr) ← disp	LI	r,disp	0 1 0 1 0 0	r	disp	
RCPY	Register Copy	(ACdr) ← (ACsr)	RCPY	sr,dr	0 1 0 1 1 1	dr	sr	not used
RXCH	Register Exchange	(ACdr) ← (ACsr), (ACsr) ← (ACdr)	RXCH	sr,dr	0 1 1 0 1 1			
XCHRS	Exchange Register and Stack	(STK) ← (ACr), (ACr) ← (STK)	XCHRS	r	0 0 0 1 1 1	r	not used	
CFR	Copy Flags Into Register	(ACr) ← (FR)	CFR	r	0 0 0 0 0 1			
CRF	Copy Register Into Flags	(FR) ← (ACr)	CRF	r	0 0 0 0 1 0			
PUSH	Push Register Onto Stack	(STK) ← (ACr)	PUSH	r	0 1 1 0 0 0			
PULL	Pull Stack Into Register	(ACr) ← (STK)	PULL	r	0 1 1 0 0 1			
PUSHF	Push Flags Onto Stack	(STK) ← (FR)	PUSHF		0 0 0 0 1 1	not used		
PULLF	Pull Stack Into Flags	(FR) ← (STK)	PULLF		0 0 0 1 0 0			
6. **Register Data Operate Instructions**								
RADD	Register Add	(ACdr) ← (ACdr) + (ACsr), OV, CY	RADD	sr,dr	0 1 1 0 1 0	dr	sr	not used
RADC	Register Add With Carry	(ACdr) ← (ACdr) + (ACsr) + (CY), OV, CY	RADC	sr,dr	0 1 1 1 0 1			
RAND	Register And	(ACdr) ← (ACdr) ∧ (ACsr).	RAND	sr,dr	0 1 0 1 0 1			
RXOR	Register Exclusive OR	(ACdr) ← (ACdr) ⊻ (ACsr)	RXOR	sr,dr	0 1 0 1 1 0			
CAI	Complement and Add Immediate	(ACr) ← ~ (ACr) + disp	CAI	r,disp	0 1 1 1 0 0	r	disp	
7. **Shift And Rotate Instructions**								
SHL	Shift Left	(ACr) ← (ACr) shifted left n places, w/wo link	SHL	r,n,ℓ	0 0 1 0 1 0	r	n	ℓ
SHR	Shift Right	(ACr) ← (ACr) shifted right n places, w/wo link	SHR	r,n,ℓ	0 0 1 0 1 1			
ROL	Rotate Left	(ACr) ← (ACr) rotated left n places, w/wo link	ROL	r,n,ℓ	0 0 1 0 0 0			
ROR	Rotate Right	(ACr) ← (ACr) rotated right n places, w/wo link	ROR	r,n,ℓ	0 0 1 0 0 1			
8. **Miscellaneous Instructions**								
HALT	Halt	Halt	HALT		0 0 0 0 0 0	not used		
SFLG	Set Flag	$(FR)_{fc}$ ← 1	SFLG	fc	0 0 1 1	fc	1	not used
PFLG	Pulse Flag	$(FR)_{fc}$ ← 1, $(FR)_{fc}$ ← 0	PFLG	fc	0 0 1 1	fc	0	

TABLE 5. Notations/Symbols Used in Instruction Descriptions

NOTATION/ SYMBOL	MEANING
ACr	Denotes specific working register (AC0, AC1, AC2, or AC3), where r is number of accumulator referenced in instruction.
cc	Denotes 4-bit condition code value for conditional branch instructions.
CRY	Indicates Carry Flag is set if carry exists due to instruction (either addition or subtraction) or reset if no carry exists.
disp	Stands for displacement value and represents operand in nonmemory-reference instruction or address field in memory-reference instruction. Disp is 8-bit, signed twos-complement number except when base page is referenced; in latter case, disp is unsigned if BPS = 0.
dr	Denotes number of destination working register specified in instruction-word field. Working register is AC0, AC1, AC2, or AC3.
EA	Denotes effective address specified by instructions directly, indirectly, or by indexing. Effective address contents are used during execution of instruction. See Table 3.
fc	Denotes number of referenced flag. **NOTE** Refer to Chapter 2, PACE Users Manual, for flag assignments.
FR	Denotes Status Flag Register.
IEN	Denotes Interrupt Enable Flag.
ℓ	Denotes inclusion of 1-bit Link (LINK) Flag in shift operations.
n	Unsigned number indicating number of bit positions to be shifted in shift and rotate instructions.
OVF	Indicates Overflow Flag is set if overflow exists due to instruction (either addition or subtraction) or is reset if no overflow exists. Overflow occurs if signs of operands are alike and sign of result is different from operands.
PC	Denotes Program Counter. During address formation, PC is incremented by 1 to contain address 1 greater than that of instruction being executed.
r	Denotes number of working register specified in instruction-word field. Working register is AC0, AC1, AC2, or AC3.
STK	Denotes top word of 10-word last-in/first-out stack.
sr	Denotes number of source working register specified in instruction-word field. Working register is AC0, AC1, AC2, or AC3.
xr	When not zero, xr value designates number of register to be used in indexed and relative memory addressing modes. When zero, base-page addressing is indicated. See Table 3.
( )	Denotes contents of item within parentheses. (ACr) is read as 'contents of ACr'. (EA) is read as 'contents of EA'.
[ ]	Denotes 'result of'.
~	Indicates logical complement (ones complement) of value on right-hand side of ~.
→	Means 'replaces'.
←	Means 'is replaced by'.
@	Appearing in operand field of instruction, denotes indirect addressing.
$^{+}10$	Modulo 10 addition.
∧	Denotes AND operation.
∨	Denotes OR operation.
∀	Denotes EXCLUSIVE-OR operation.

The **BRANCH INSTRUCTIONS** group consists of the seven following instructions: BOC, JMP, JMP@, JSR, JSR@, RTI, and RTS.

NOTE: JMP@ and JSR@ are specified to the Assembler as JMP and JSR with indirection specified by the address field.

Six of the seven instructions (excepting BOC) address memory and peripheral devices, and each is described as follows:

- Name of instruction followed by mnemonic in parentheses
- Binary instruction format
- Operation in equation notation
- Assembly language instruction format (see "Assembler" chapter, PACE Users Manual, for further information)
- Description of operation

BRANCH ON CONDITION (BOC)

15 12	11 8	7 0
0 1 0 0	cc	disp

Operation: (PC) ← (PC) + disp (sign extended) if condition is true.

Format: BOC cc, disp

Description: There are 16 possible condition codes (cc). The condition codes are listed in Table 6. If the condition for branching designated by cc is true, the value of disp (sign extended from bit 7 through bit 15) is added to PC and the sum is stored in PC.

NOTE: PC addresses the location following the BOC when the addition occurs (that is, the branch is relative to the next instruction after BOC).

The initial contents of PC are lost. Program control is transferred to the location specified by the contents of the new PC.

TABLE 6. Branch Conditions

CONDITION CODE (cc)	MNEMONIC	CONDITION
0000	STFL	Stack full.
0001	REQ0	(AC0) equal to zero (1).
0010	PSIGN	(AC0) has positive sign (2).
0011	BIT0	Bit 0 of AC0 true.
0100	BIT1	Bit 1 of AC0 true.
0101	NREQ0	(AC0) is nonzero (1).
0110	BIT2	Bit 2 of AC0 is true.
0111	CONTIN	CONTIN (continue) input is true.
1000	LINK	LINK is true.
1001	IEN	IEN is true.
1010	CARRY	CARRY is true.
1011	NSIGN	(AC0) has negative sign (2).
1100	OVF	OVF is true.
1101	JC13	JC13 input is true (3).
1110	JC14	JC14 input is true.
1111	JC15	JC15 input is true.

Note 1: If selected data length is 8 bits, only bits 0 through 7 of AC0 are tested.
Note 2: Bit 7 is sign bit (instead of bit 15) if selected data length is 8 bits.
Note 3: JC13 is used by PACE Microprocessor Development System and is not accessible during prototyping.

JUMP (JMP)

15 10	9 8	7 0
0 0 0 1 1 0	xr	disp

Operation: (PC) ← EA

Format: JMP disp (xr)

Description: The effective address EA replaces the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

JUMP INDIRECT (JMP@)

15 10	9 8	7 0
1 0 0 1 1 0	xr	disp

Operation: (PC) ← (EA)

Format: JMP@ @disp (xr)

Description: The contents of the effective address replace the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

JUMP TO SUBROUTINE (JSR)

15 10	9 8	7 0
0 0 0 1 0 1	xr	disp

Operation: (STK) ← (PC), (PC) ← EA

Format: JSR disp (xr)

Description: The contents of PC are stored in the top of the stack. The effective address replaces the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

JUMP TO SUBROUTINE INDIRECT (JSR@)

15 10	9 8	7 0
1 0 0 1 0 1	xr	disp

Operation: (STK) ← (PC), (PC) ← (EA)

Format: JSR @disp (xr)

Description: The contents of PC are stored in the top of the stack. The contents of the effective address replace the contents of PC. The next instruction is fetched from the location designated by the new contents of PC.

RETURN FROM SUBROUTINE (RTS)

15 10	9 8	7 0
1 0 0 0 0 0	Not Used	disp

Operation: (PC) ← (STK) + disp (sign extended)

Format: RTS disp

Description: The contents of PC are replaced by disp added to the contents pulled from the top of the stack. Program control is transferred to the location specified by the new contents of PC.

NOTE: RTS is used primarily to return from subroutines entered by JSR.

RETURN FROM INTERRUPT (RTI)

15 10	9 8	7 0
0 1 1 1 1 1	Not Used	disp

Operation: (PC) ← (STK) + disp (sign extended), IEN = 1

Format: RTI disp

Description: The Interrupt Enable Flag (IEN) is set. The contents of PC are replaced by disp added to the contents pulled from the top of the stack. Program control is transferred to the location specified by the new contents of PC.

NOTE: RTI is used primarily to exit from an interrupt routine.

Six **SKIP INSTRUCTIONS** are provided: SKNE, SKG, SKAZ, AISZ, ISZ, and DSZ.

SKIP IF NOT EQUAL (SKNE)

15 12	11 10	9 8	7 0
1 1 1 1	r	xr	disp

Operation: If (ACr) ≠ (EA), (PC) ← (PC) + 1

Format: SKNE r, disp (xr)

Description: The contents of ACr and the contents of the effective memory location EA are compared. If the contents of ACr and the effective memory location EA are not equal, the next instruction in sequence is skipped. The contents of ACr and EA are unaltered. If an 8-bit data length is selected, only the lower 8 bits are compared.

SKIP IF GREATER (SKG)

15 10	9 8	7 0
1 0 0 1 1 1	xr	disp

Operation: If (AC0) > (EA), (PC) ← (PC) + 1

Format: SKG 0,disp (xr)

Description: The contents of AC0 and the contents of the effective memory location EA are compared as signed numbers. If the contents of AC0 are greater (more positive) than the contents of EA, the next instruction in sequence is skipped. The contents of AC0 and EA are unaltered.

NOTE: The comparison is performed by subtraction. If an 8-bit data length is selected, only the lower 8 bits are compared.

SKIP IF AND IS ZERO (SKAZ)

15 10	9 8	7 0
1 0 1 1 1 0	xr	disp

Operation: If [(AC0) $\wedge$ (EA)] = 0, (PC) ← (PC) + 1

Format: SKAZ 0,disp (xr)

Description: The contents of AC0 and the contents of the effective memory location EA are ANDed. If the result equals zero, the next instruction in sequence is skipped. The contents of AC0 and EA are unaltered. If an 8-bit data length is selected, only the lower 8 bits are tested.

INCREMENT AND SKIP IF ZERO (ISZ)

15 10	9 8	7 0
1 0 0 0 1 1	xr	disp

Operation: (EA) ← (EA) +1; if (EA) = 0, (PC) ← (PC) + 1

Format: ISZ disp (xr)

Description: The contents of EA are incremented by one. If the new contents of EA equal zero, the next instruction in sequence is skipped. If an 8-bit data length is selected, only the lower 8 bits are tested.

DECREMENT AND SKIP IF ZERO (DSZ)

15 10	9 8	7 0
1 0 1 0 1 1	xr	disp

Operation: (EA) ← (EA) – 1; if (EA) = 0, (PC) ← (PC) + 1

Format: DSZ disp (xr)

Description: The contents of EA are decremented by one. If the new contents of EA equal zero, the next instruction in sequence is skipped. If an 8-bit data length is selected, only the lower 8 bits are tested.

ADD IMMEDIATE, SKIP IF ZERO (AISZ)

15 10	9 8	7 0
0 1 1 1 1 0	r	disp

Operation: (ACr) ← (ACr) + disp (sign extended). If new (ACr) = 0, (PC) ← (PC) + 1

Format: AISZ r,disp

Description: The contents of Register ACr are replaced by the sum of the contents of ACr and disp (sign bit 7 extended through bit 15). The initial contents of ACr are lost. If the new contents of ACr equal zero, the contents of PC are incremented by one, thus skipping the next instruction. The AISZ Instruction always tests the full 16-bit result independent of the data length selected.

NOTE: Testing the 16-bit result in conjunction with no change to the status indicators allows AISZ to conveniently used for modifying 16-bit index values while working with 8-bit data.

The five **MEMORY DATA-TRANSFER INSTRUCTIONS** (LD, LD@, ST, ST@, and LSEX) effect data transfers between the registers and memory or peripheral devices.

LOAD (LD)

15 12	11 10	9 8	7 0
1 1 0 0	r	xr	disp

Operation: (ACr) ← (EA)

Format: LD r,disp (xr)

Description: The contents of ACr are replaced by the contents of EA. The initial contents of ACr are lost; the contents of EA are unaltered.

LOAD INDIRECT (LD@)

15 10	9 8	7 0
1 0 1 0 0 0	xr	disp

Operation: (AC0) ← ((EA))

Format: LD 0,@disp (xr)

Description: The contents of AC0 are replaced indirectly by the contents of EA. The initial contents of AC0 are lost; the contents of EA and the location that designates EA are unaltered.

STORE (ST)

15 12	11 10	9 8	7 0
1 1 0 1	r	xr	disp

Operation: (EA) ← (ACr)

Format: ST r,disp (xr)

Description: The contents of EA are replaced by the contents of ACr. The initial contents of EA are lost; the contents of ACr are unaltered.

STORE INDIRECT (ST@)

15 10	9 8	7 0
1 0 1 1 0 0	xr	disp

Operation: ((EA)) ← (AC0)

Format: ST 0,@disp (xr)

Description: The contents of EA are replaced indirectly by the contents of AC0. The initial contents of EA are lost; the contents of AC0 and the location that designates EA are unaltered.

LOAD WITH SIGN EXTENDED (LSEX)

15 ... 10	9 8	7 ... 0
1 0 1 1 1 1	xr	disp

Operation: (AC0) ← (EA) (sign extended)

Format: LSEX 0,disp (xr)

Description: The contents of AC0 are replaced by the contents of EA with bit 7 extended through bits 8 through 15. The initial contents of AC0 are lost; the contents of EA are unaltered.

NOTE: The LSEX Instruction allows 8-bit arithmetic data to be loaded from an 8-bit data memory or peripheral device register and to be operated on as 16-bit arithmetic data.

The five **MEMORY DATA-OPERATE INSTRUCTIONS** (AND, OR, ADD, DECA, and SUBB) provide the memory-register operations.

AND (AND)

15 ... 10	9 8	7 ... 0
1 0 1 0 1 0	xr	disp

Operation: (AC0) ← (AC0) ∧ (EA)

Format: AND 0,disp (xr)

Description: The contents of Accumulator AC0 and the contents of the effective memory location EA are ANDed, and the result is stored in AC0. The initial contents of AC0 are lost, and the contents of EA are unaltered.

OR (OR)

15 ... 10	9 8	7 ... 0
1 0 1 0 0 1	xr	disp

Operation: (AC0) ← (AC0) ∧ (EA)

Format: OR 0,disp (xr)

Description: The contents of Accumulator AC0 and the contents of the effective memory location EA are ORed inclusively. The result is stored in AC0. The initial contents of AC0 are lost, and the contents of EA are unaltered.

ADD (ADD)

15 ... 12	11 10	9 8	7 ... 0
1 1 1 0	r	xr	disp

Operation: (ACr) ← (ACr) + (EA), OVF, CRY

Format: ADD r,disp (xr)

Description: The contents of ACr are added algebraically to the contents of the effective memory location EA. The sum is stored in ACr, and the contents of EA are unaltered. The initial contents of ACr are lost. The Overflow or Carry Flag is set if overflow or carry occurs, respectively; otherwise the Overflow and Carry Flags are cleared.

SUBTRACT WITH BORROW (SUBB)

15 ... 10	9 8	7 ... 0
1 0 0 1 0 0	xr	disp

Operation: (AC0) ← (AC0) + ~ (EA) + (CRY), OVF, CRY

Format: SUBB 0,disp (xr)

Description: The contents of AC0 are added to the complement of the effective memory location EA and the carry. The result is stored in AC0, and the contents of EA are unaltered. The initial contents of AC0 are lost. The Carry and Overflow Flags are set according to the result of the operation.

NOTE: The carry input should be set true for single-word operations and serves as a borrow for multiple-word operations.

DECIMAL ADD (DECA)

15 ... 10	9 8	7 ... 0
1 0 0 0 1 0	xr	disp

Operation: (AC0) ← (AC0) $+_{10}$ (EA) $+_{10}$ (CRY),OVF, CRY

Format: DECA 0,disp (xr)

Description: The contents of Register AC0 are treated as a 4-digit number and added modulo 10 (for each digit) to the contents of memory location EA (treated as a 4-digit number) and the carry. The initial contents of AC0 are lost; the contents of EA are unaltered. The Carry Flag is set based on a decimal carry output. The Overflow Flag is set to an arbitrary state.

NOTE: Subtraction may be performed by forming the tens complement and using the DECA Instruction.

Ten **REGISTER DATA-TRANSFER INSTRUCTIONS** are provided as follows: LI, RCPY, RXCH, XCHRS, CFR, CRF, PUSH, PULL, PUSHF, PULLF. Register data-transfer instructions effect data transfers among the registers, flags and stack.

LOAD IMMEDIATE (LI)

15					10	9	8	7							0
0	1	0	1	0	0	r		disp							

Operation: (ACr) ← disp (sign extended)

Format: LI r,disp

Description: The contents of Accumulator ACr are replaced by disp with sign bit 7 extended through bit 15. The initial contents of ACr are lost.

REGISTER COPY (RCPY)

15					10	9	8	7	6	5					0
0	1	0	1	1	1	dr		sr		Not Used					

Operation: (ACdr) ← (ACsr)

Format: RCPY sr, dr

Description: The contents of the Destination Register ACdr are replaced by the contents of the Source Register ACsr. The initial contents of ACdr are lost, and the initial contents of ACsr are unaltered.

REGISTER EXCHANGE (RXCH)

15					10	9	8	7	6	5					0
0	1	1	0	1	1	dr		sr		Not Used					

Operation: (ACsr) ← (ACdr), (ACdr) ← (ACsr)

Format: RXCH sr, dr

Description: The contents of Source Register ACsr and Destination Register ACdr are exchanged.

EXCHANGE REGISTER AND STACK (XCHRS)

15					10	9	8	7							0
0	0	0	1	1	1	r		Not Used							

Operation: (STK) ← (ACr), (ACr) ← (STK)

Format: XCHRS r

Description: The contents of the top of the stack and the register designated by ACr are exchanged.

NOTE: The XCHRS Instruction provides a convenient means of placing a subroutine return address into an index register for modification and/or use to pass parameters.

COPY FLAGS TO REGISTER (CFR)

15					10	9	8	7							0
0	0	0	0	0	1	r		Not Used							

Operation: (ACr) ← (FR)

Format: CFR r

Description: The contents of Accumulator ACr are replaced by the contents of the Flag Register (FR). The initial contents of ACr are lost; the contents of FR are unaltered.

COPY REGISTER TO FLAGS (CRF)

15					10	9	8	7							0
0	0	0	0	1	0	r		Not Used							

Operation: (FR) ← (ACr)

Format: CRF r

Description: The contents of FR are replaced by the contents of Accumulator ACr. The initial contents of FR are lost; the contents of ACr are unaltered.

PUSH ONTO STACK (PUSH)

15					10	9	8	7							0
0	1	1	0	0	0	r		Not Used							

Operation: (STK) ← (ACr)

Format: PUSH r

Description: The stack is pushed by the contents of the accumulator designated by ACr. Thus, the top of the stack holds the contents of ACr, and the stack pointer is incremented by one. The initial contents of ACr are unaltered.

NOTE: If PUSH causes the stack pointer to go to 1000_2 (8_{10}; that is, nine words on stack) the Stack-full Interrupt request is set.

PULL FROM STACK (PULL)

15					10	9	8	7							0
0	1	1	0	0	1	r		Not Used							

Operation: (ACr) ← (STK)

Format: PULL r

Description: The stack is pulled. The contents from the top of the stack replace the contents of the Accumulator ACr. The initial contents of ACr are lost. The contents of the stack pointer are decremented by one.

NOTE: If the stack pointer goes to −1 (that is, no words left on stack) a Stack-empty Interrupt request is generated.

PUSH FLAG REGISTER ONTO STACK (PUSHF)

15 10	9 0
0 0 0 0 1 1	Not Used

Operation: (STK) ← (FR)

Format: PUSHF

Description: The contents of FR are pushed onto the stack. The contents of FR are unchanged.

PULL FLAG REGISTER FROM STACK (PULLF)

15 10	9 0
0 0 0 1 0 0	Not Used

Operation: (FR) ← (STK)

Format: PULLF

Description: The contents of FR are replaced by the contents pulled from the top of the stack. The initial contents of FR are lost.

The five **REGISTER DATA-OPERATE INSTRUCTIONS** (RADD, RADC, RAND, RXOR, and CAI) allow modification of register data.

REGISTER ADD (RADD)

15 10	9 8	7 6	5 0
0 1 1 0 1 0	dr	sr	Not Used

Operation: (ACdr) ← (ACsr) + (ACdr), OVF, CRY

Format: RADD sr, dr

Description: The contents of the Destination Register ACdr are replaced by the sum of the contents of ACdr and the Source Register ACsr. The initial contents of ACdr are lost, and the contents of ACsr are unaltered. The Overflow and Carry Flags are modified according to the result.

REGISTER ADD WITH CARRY (RADC)

15 10	9 8	7 6	5 0
0 1 1 1 0 1	dr	sr	Not Used

Operation: (ACdr) ← (ACdr) + (ACsr) + (CRY), OVF, CRY

Format: RADC sr, dr

Description: The contents of the Destination Register ACdr are replaced by the sum of the contents of ACdr and the Source Register ACsr and the carry. The initial contents of ACdr are lost, and the contents of ACsr are unaltered. The Overflow and Carry Flags are modified according to the result.

REGISTER AND (RAND)

15 10	9 8	7 6	5 0
0 1 0 1 0 1	dr	sr	Not Used

Operation: (ACdr) ← (ACdr) ∨ (ACsr)

Format: RAND sr, dr

Description: The contents of the Destination Register ACdr are replaced by the result of ANDing the contents of ACdr and the contents of the Source Register ACsr. The initial contents of ACdr are lost, and the initial contents of ACsr are unaltered.

REGISTER EXCLUSIVE-OR (RXOR)

15 10	9 8	7 6	5 0
0 1 0 1 1 0	dr	sr	Not Used

Operation: (ACdr) ← (ACdr) ⊻ (ACsr)

Format: RXOR sr, dr

Description: The contents of the Destination Register ACdr are replaced by the result of exclusively ORing the contents of ACdr and the contents of the Source Register ACsr. The initial contents of ACdr are lost, and the initial contents of ACsr are unaltered.

COMPLEMENT AND ADD IMMEDIATE (CAI)

15 10	9 8	7 0
0 1 1 1 0 0	r	disp

Operation: (ACr) ← ~ (ACr) + disp (sign extended)

Format: CAI r, disp

Description: The contents of Accumulator ACr are replaced by the sum of the complement of ACr and disp (sign bit 7 extended through bit 15). The initial contents of ACr are lost.

NOTE: Values of zero and one in the disp field produce the ones and twos complement, respectively, of (ACr).

The four **SHIFT AND ROTATE INSTRUCTIONS** (SHL, SHR, ROL, and ROR) are described in the following paragraphs.

SHIFT LEFT (SHL)

15 10	9 8	7 1	0
0 0 1 0 1 0	r	n	ℓ

Operation: (ACr) ← (ACr) shifted left n places, include LINK if ℓ = 1, $(ACr)_{8:15}$ ← 0 if data length = 8 bits

Format: SHL r, n, ℓ

Description: The contents of Register ACr are shifted left n (n = 0 – 127) bit positions. If the selected data length is 8 bits, then bits 8 through 15 are set to zero. Data shifted out of the most significant bit for the specified data length are lost if ℓ = 0 and are loaded into the LINK if ℓ = 1. A schematic representation of the various SHL Instruction possibilities is shown in *Figure 11.*

SHIFT RIGHT (SHR)

15 10	9 8	7 1	0
0 0 1 0 1 1	r	n	ℓ

Operation: (ACr) ← (ACr) shifted right n places, include LINK if ℓ = 1, $(ACr)_{8:15}$ ← 0 if data length = 8 bits

Format: SHR r, n, ℓ

Description: The contents of Register ACr are shifted right n (n = 0 – 127) bit positions. If the selected data length is 8 bits, then bits 8 through 15 are set to zero. Zeroes are shifted into the most significant bit for the specified data length if ℓ = 0. The contents of the LINK are shifted in if ℓ = 1, and the contents of the LINK are unchanged. Data shifted out of the least significant bit are lost. A schematic representation of the various SHR Instruction possibilities is shown in *Figure 12.*

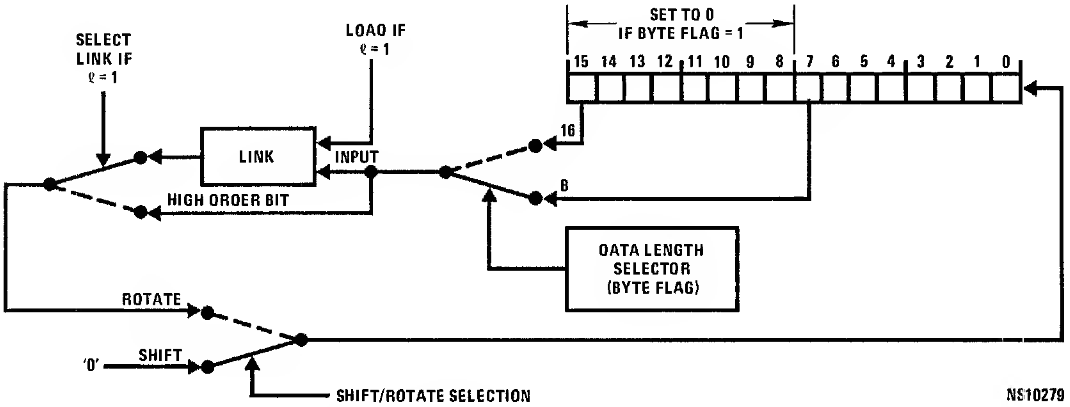


FIGURE 11. Left Shift and Rotate Instructions

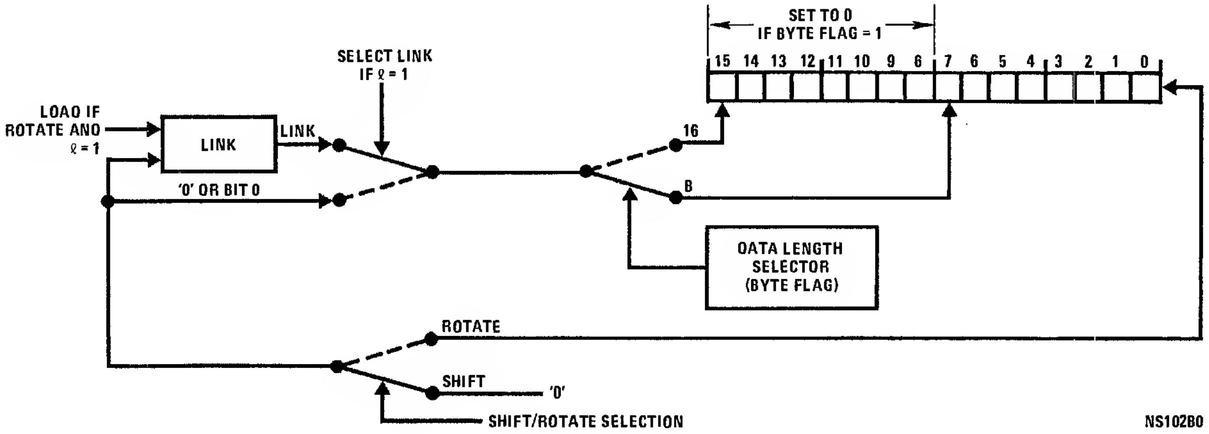


FIGURE 12. Right Shift and Rotate Instructions

ROTATE LEFT (ROL)

15 10	9 8	7 1	0
0 0 1 0 0 0	r	n	ℓ

Operation: $(ACr) \leftarrow (ACr)$ rotated left n places, include LINK if $\ell = 1$, $(ACr)_{8:15} \leftarrow 0$ if data length = 8 bits

Format: ROL r, n, ℓ

Description: The contents of Register ACr are rotated left n (n = 0 – 127) bit positions. If the selected data length is 8 bits, then bits 8 through 15 are set to zero. Data shifted out of the most significant bit position for the specified data length are shifted into the least significant bit if $\ell = 0$, and into the LINK if $\ell = 1$, in which case the least significant bit is loaded from the LINK. A schematic representation of the various ROL Instruction possibilities is shown in *Figure 11.*

ROTATE RIGHT (ROR)

15 10	9 8	7 1	0
0 0 1 0 0 1	r	n	ℓ

Operation: $(ACr) \leftarrow (ACr)$ rotated right n places, include LINK if $\ell = 1$. $(ACr)_{8:15} \leftarrow 0$ if data length = 8 bits

Format: ROR r, n, ℓ

Description: The contents of Register ACr are rotated right n (n = 0 – 127) bit positions. If the selected data length is 8 bits, then bits 8 through 15 are set to zero. Data shifted out of the least significant bit are shifted into the most significant bit for the specified data length if $\ell = 0$, and into the LINK if $\ell = 1$, in which case the most significant bit is loaded from the LINK. A schematic representation of the various ROR Instruction possibilities is shown in *Figure 12.*

The three **MISCELLANEOUS INSTRUCTIONS** are HALT, SFLG, and PFLG.

HALT (HALT)

15 10	9 0
0 0 0 0 0 0	Not Used

Format: HALT

Description: The processor halts and remains halted until the CONTINUE jump condition input makes a transition from logic '1' to logic '0'.

NOTE: CONTINUE must be held at logic one for at least four clock cycles prior to the transition and must then be held at logic zero for at least four clock cycles.

SET FLAG (SFLG)

15 12	11 8	7	6 0
0 0 1 1	fc	1	Not Used

Operation: $(FR)_{fc} \leftarrow 1$

Format: SFLG fc

Description: The flag, or bit of FR, specified by flag code fc is set true. All other bits of FR are unaltered.

NOTE: The functions of the bits in the Status Flag Register are defined in Chapter 2, PACE Users Manual.

PULSE FLAG (PFLG)

15 12	11 8	7	6 0
0 0 1 1	fc	0	Not Used

Operation: $(FR)_{fc} \leftarrow 1, (FR)_{fc} \leftarrow 0$

Format: PFLG fc

Description: The flag (bit fc of FR) is first set true and then set false (after four clock periods), causing a pulsing or resetting of the flag, depending on the initial state of the flag. All other bits of FR are unaffected.

NOTE: Operation code 1000 01 is unused—causes JMP PC ± disp. Operation code 1011 01 is unused—causes SKIP if scratch register 1=0.

The formulas for computing the execution times of PACE instructions are listed in Table 7. The formulas are presented in terms of machine (microinstruction) cycles (M) and I/O data-transfer cycle extends (E_R for read and E_W for write). Each machine cycle (M) consists of four clock cycles. The following example shows the method of calculating the instruction execution times.

EXAMPLE: The formula (listed in Table 7) for the execution time of a RADD Instruction is $4M + E_R$. If the clock cycle (or period) is 500 nanoseconds and the read cycle extend is 500 nanoseconds, then: $M = 4(0.5\mu s) = 2\mu s$; $E_R = 0.5\mu s$; therefore: $4M+E_R=4(2\mu s)+0.5\mu s=8.5\mu s$. Thus, under the hypothetical clock cycle and read cycle extend times used, the RADD Instruction requires 8.5 microseconds for execution.

TABLE 7. Instruction Execution Times

MNEMONIC	MEANING	EXECUTION TIME FORMULA
BRANCH INSTRUCTIONS		
BOC	Branch On Condition	$5M + E_R$ + 1M if branch
JMP	Jump	$4M + E_R$
JMP@	Jump Indirect	$4M + 2E_R$
JSR	Jump to Subroutine	$5M + E_R$
JSR@	Jump to Subroutine Indirect	$5M + 2E_R$
RTS	Return from Subroutine	$5M + E_R$
RTI	Return from Interrupt	$6M + E_R$
SKIP INSTRUCTIONS		
SKNE	Skip if Not Equal	$5M + 2E_R$ + 1M if skip
SKG	Skip if Greater	$7M + 2E_R$ + 1M if skip
SKAZ	Skip if AND is Zero	$5M + 2E_R$ + 1M if skip
ISZ	Increment and Skip if Zero	$7M + 2E_R + E_W$ + 1M if skip
DSZ	Decrement and Skip if Zero	$7M + 2E_R + E_W$ + 1M if skip
AISZ	Add Immediate, Skip if Zero	$5M + E_R$ + 1M if skip
MEMORY DATA-TRANSFER INSTRUCTIONS		
LD	Load	$4M + 2E_R$
LD@	Load Indirect	$5M + 3E_R$
ST	Store	$4M + E_R + E_W$
ST@	Store Indirect	$4M + 2E_R + E_W$
LSEX	Load with Sign Extended	$4M + 2E_R$
MEMORY DATA-OPERATE INSTRUCTIONS		
AND	AND	$4M + 2E_R$
OR	OR	$4M + 2E_R$
ADD	Add	$4M + 2E_R$
SUBB	Subtract with Borrow	$4M + 2E_R$
DECA	Decimal Add	$7M + 2E_R$
REGISTER DATA-TRANSFER INSTRUCTIONS		
LI	Load Immediate	$4M + E_R$
RCPY	Register Copy	$4M + E_R$
RXCH	Register Exchange	$6M + E_R$
XCHRS	Exchange Register and Stack	$6M + E_R$
CFR	Copy Flags into Register	$4M + E_R$
CRF	Copy Register into Flags	$4M + E_R$
PUSH	Push Register onto Stack	$4M + E_R$
PULL	Pull Stack into Register	$4M + E_R$
PUSHF	Push Flags onto Stack	$4M + E_R$
PULLF	Pull Stack into Flags	$4M + E_R$
REGISTER DATA-OPERATE INSTRUCTIONS		
RADD	Register Add	$4M + E_R$
RADC	Register Add with Carry	$4M + E_R$
RAND	Register AND	$4M + E_R$
RXOR	Register EXCLUSIVE-OR	$4M + E_R$
CAI	Complement and Add Immediate	$5M + E_R$
SHIFT AND ROTATE INSTRUCTIONS		
SHL	Shift Left	$(5 + 3n) M + E_R$, $n = 1\text{-}127$; $6M + E_R$, $n = 0$
SHR	Shift Right	$(5 + 3n) M + E_R$, $n = 1\text{-}127$; $6M + E_R$, $n = 0$
ROL	Rotate Left	$(5 + 3n) M + E_R$, $n = 1\text{-}127$; $6M + E_R$, $n = 0$
ROR	Rotate Right	$(5 + 3n) M + E_R$, $n = 1\text{-}127$; $6M + E_R$, $n = 0$
MISCELLANEOUS INSTRUCTIONS		
HALT	Halt	- - - - -
SFLG	Set Flag	$5M + E_R$
PFLG	Pulse Flag	$6M + E_R$

NOTES: M = Machine cycle time = 4 clock periods
n = Number of shifts
E_R = Extend time for read cycle
E_W = Extend time for write cycle
External interrupt response time is $7M + E_R$ plus time to finish current instruction.

The following paragraphs contain example programs that demonstrate the use of PACE instructions. Refer to Chapter 10, PACE Users Manual, for a description of the program listing format.

The decimal addition program (see Table 8) adds two 16-digit BCD strings that are packed four digits per word. The two strings to be added are stored in memory starting at locations STR1 and STR2. The resulting digit string is stored in memory starting at location STR2.

Representation of negative decimal numbers in tens-complement form may be desirable for many PACE applications, since the Decimal-Add Instruction can then be used directly for signed number additions. The tens-complement program converts an unsigned BCD number to a tens-complement negative number representation.

The sign of a tens-complement number can be tested by using the BOC Instruction with the PSIGN jump condition to test the most significant word of the decimal number.

NOTE: Negative numbers have leading nines while positive numbers have leading zeroes.

The tens-complement program presented in Table 9 converts a 16-digit number packed in four words of memory beginning at location NUM.

TABLE 8. Decimal Addition Program Example

```
ADDR1:   .WORD   STR1       ;Address of addend string
ADDR2:   .WORD   STR2       ;Address of augend/result string
START:   LI      R1,4       ;Number digits/4 to AC1 (loop count)
         LD      R2,ADDR1   ;Load index registers with
         LD      R3,ADDR2   ; argument addresses
         PFLG    CY         ;Clear Carry Flags
LOOP:    LD      R0, (R2)   ;Addend to AC0
         DECA    R0, (R3)   ;Decimal add with augend
         ST      R0, (R3)   ;Store result
         AISZ    R2,1       ;Increment index
         AISZ    R3,1       ;  registers
         AISZ    R1,-1      ;Decrement loop count
         JMP     LOOP       ;Add next word
```

NOTE: Execution time = $155M + 42E_R + 4E_W = 310\mu s$ for 500 ns clock.

TABLE 9. Tens-Complement Program Example

```
ADDR:    .WORD   NUM         ;Decimal string address
CONST:   .WORD   X'999A      ;Constant
START:   LI      R1,4        ;Loop count to AC1
         LD      R2,ADDR     ;Address to AC2 index register
         SFLG    CRY         ;Set Carry Flag for first loop
LOOP     LD      R0,CONST    ;Constant to AC0
         SUBB    R0, (R2)    ;Complement and add decimal
                             ;  number plus carry
         ST      R0, (R2)    ;Store result
         PFLG    CRY         ;Clear carry for subsequent loop
         AISZ    R2, 1       ;Increment pointer
         AISZ    R1,-1       ;Decrement loop count
         JMP     LOOP        ;Repeat loop
```

The decimal subtraction program listed in Table 10 performs a decimal subtract by forming the tens complement and using the Decimal-Add Instruction. The 16-digit string, starting at location STR2, is subtracted from the string starting at location STR1.

Two binary-multiplication program examples are provided in Table 11. The first program example multiplies the 16-bit value in AC2 by the 16-bit value in AC0 and provides a 32-bit result in AC1 (high order) and AC0 (low order).

NOTE: Positive numbers of 16-bit magnitude are assumed (that is, most significant bit is zero).

The second program multiplies the 16-bit value in AC2 by the 16-bit value in AC0 and provides a 32-bit result in AC0 (high order) and AC1 (low order).

NOTE: 16-bit magnitude only is assumed.

TABLE 10. Decimal Subtraction Program Example

```
ADDR1:   .WORD   STR1        ;Decimal string addresses
ADDR2:   .WORD   STR2        ;
CONST:   .WORD   X'9999      ;Tens complement constant
START:   LI      R1,4        ;Loop count to AC1
         LD      R2,ADDR1    ;Decimal addresses to index registers
         LD      R3,ADDR2    ;
         SFLG    CY          ;Set carry in for L.S. digit tens complement
LOOP:    LD      R0,CONST    ;Form nines complement of number at STR2
         SUBB    R0, (R3)    ;  carry set true to form tens complement
         DECA    R0, (R2)    ;Decimal add
         ST      R0, (R3)    ;Save result
         AISZ    R2,1        ;Increment address
         AISZ    R3,1        ;Increment address
         AISZ    R1,-1       ;Decrement loop count
         JMP     LOOP        ;Repeat loop
```

NOTE: Execution time = 170M + 50E_R + 4E_W = 340μs for 500 ns clock.

TABLE 11. Binary-Multiplication Program Examples

```
START:          LI          R1,0            ;Clear result register
                LI          R3,16           ;Loop count to AC3
                CAI         R0,0            ;Complement multiplier
LOOP:           BOC         BIT0, SHIFT     ;Test bit zero
                RADD        R2,R1           ;Add multiplicand to result
SHIFT:          PFLG        LINK            ;Clear link
                ROR         R1,1,1          ;Shift AC1 into link
                SHR         R0,1,1          ;Shift link into AC0
                AISZ        R3,-1           ;Decrement loop count
                JMP         LOOP            ;Repeat loop
```

NOTE: Execution time = 634M + 114E_R = 1268μs, maximum, for 500 ns clock.

```
CONST:          .WORD       X'FFFF          ;Constant for double-precision addition
START:          LI          R1,0            ;Clear result register
                LI          R3,16           ;Loop count to AC3
                CAI         R0,0            ;Complement multiplier
LOOP:           RADD        R1,R1           ;Shift result left into carry
                RADC        R0,R0           ;Shift carry into multiplier and multiplier
                                            ;  into carry
                BOC         CARRY, TEST     ;Test for add
                RADD        R2,R1           ;Add multiplicand to result
                SUBB        R0,CONST        ;Add carry to high-order result
TEST:           AISZ        R3, -1          ;Decrement loop count
                JMP         LOOP            ;Repeat loop
```

NOTE: Execution time = 474M + 130E_R = 948μs, maiximum, for 500 ns clock.

Stack Service Routine

The Stack Service Routine listed in Table 12 pushes four words onto, or pulls four words from, the software stack when the hardware stack is full or empty, respectively. Thus, successive interrupts are prevented when a push instruction is followed by a pull instruction; that is, the Stack Service Routine provides hysteresis.

NOTE: At least one word always should be left on the hardware stack by the Stack Service Routine to prevent a Stack-empty Interrupt from occurring after pushing the software stack. Similarly, only eight words should be pushed onto the hardware stack to prevent a Stack-full Interrupt.

The Stack Service Routine does not check for software stack overflow or underflow.

TABLE 12. Stack Service Routine

```
                      .TITLE  STKINT,' SOFTWARE STACK'
                      .LOCAL
              ;
              ;   STKINT MAINTAINS A SOFTWARE STACK BY EMPTYING AND FILLING
              ;   THE HARDWARE STACK WHENEVER A STACK INTERRUPT OCCURS. IT
              ;   REMOVES OR REPLACES 4 WORDS AT A TIME TO MINIMIZE INTERRUPTS.
              ;
     0000     R0      =       0                ; REGISTER 0
     0001     R1      =       1                ; REGISTER 1
     0002     P2      =       2                ; REGISTER 2
     0000     STFL    =       0                ; STACK FULL CONDITION
     0001     IEN1    =       1                ; STACK INT ENABLE FLAG
     0000             .ASSCT
     0002             .=2
0002 1400 T           .WORD   STKINT
     0000             .TSECT
     1400             .=.+0]400
              ;
              ;   SAVE REGS AND DETERMINS WHETHER STACK FULL OR EMPTY.
              ;
1400 D127 T   STKINT: ST      R0,$SAV0         ; SAVE REG 0
1401 D527 T           ST      R1,$SAV1         ; SAVE REG 1
1402 D927 T           ST      R2,$SAV2         ; SAVE REG 2
1403 6400 A           PULL    R0               ; FETCH RETURN ADDRESS
1404 D126 T           ST      R0,$RETA         ; SAVE
1405 5104 A           LI      R1,4             ; NUMBER OF WORDS TO PROCESS
1406 400D A           BOC     STFL,$FULL       ; CHECK CONDITION
              ;
              ;   STACK EMPTY.  RESTORE FOUR WORDS.
              ;
1407 AD24 T   $EMP:   DSZ     $SPTR            ; ADJUST STACK POINTER
1408 A123 T           LD      R0,@$SPTR        ; LOAD WORD
1409 6000 A           PUSH    R0               ; PUSH ONTO HARDWARE STACK
140A 79FF A           AISZ    R1,-1            ; CHECK IF FINISHED
140B 19FB T           JMP     $EMP             ; GET NEXT WORD
              ;
              ;   RESTORE REGISTERS AND RETURN FROM INTERRUPT
              ;
140C C11E T   $REST:  LD      R0,$RETA         ; FETCH RETURN ADDRESS
140D 6000 A           PUSH    R0               ; RESTORE INTO STACK
140E C119 T           LD      R0,$SAV0         ; RESTORE REG 0
140F C519 T           LD      R1,$SAV1         ; RESTORE REG 1
1410 C919 T           LD      R2,$SAV2         ; RESTORE REG 2
1411 3100 A           PFLG    IEN1             ; CLEAR INTERRUPT
1412 3180 A           SFLG    IEN1             ; RE-ENABLE STACK INT
1413 7C00 A           RTI                      ; RETURN, SET INTERRUPT ENABLE

              ;
              ;   STACK FULL. FIRST SAVE TOP FIVE ELEMENTS OF STACK.
              ;
1414 C91D T   $FULL:  LD      R2,$ADR          ; ADDPESS TO STORE 5 ELEMENTS
1415 7901 A           AISZ    R1,1             ; MUST PROCESS FIVE ELEMENTS
1416 6400 A   $LP1:   PULL    R0               ; FETCH WORD FROM STACK
1417 D200 A           ST      R0,(R2)          ; STORE IN TEMPORARY LOCATION
1418 7A01 A           AISZ    R2,1             ; NEXT TEMPORARY LOCATION
1419 79FF A           AISZ    R1,-1            ; CHECK IF FINISHED
141A 19FB T           JMP     $LP1             ; GET NEXT WORD
              ;
              ;   NOW PUT BOTTOM FOUR WORDS ONTO SOFTWARE STACK
              ;
1418 5104 A           LI      R1,4             ; NUMBER OF WORDS TO REMOVE
14IC 6400 A   $LP2:   PULL    R0               ; FETCH WORD FROM STACK
141D B10E T           ST      R0,@$SPTR        ; STORE IN SOFTWARE STACK
141E 8D0D T           ISZ     $SPTR            ; INCREMENT STACK POINTER
141F 79FF A           AISZ    R1,-1            ; CHECK IF FINISHED
1420 19FB T           JMP     $LP2             ; GET NEXT WORD
              ;
              ;   FINALLY RESTORE TOP 5 WORDS TO BOTTOM OF STACK
              ;
1421 5105 A           LI      R1,5             ; NUMBER OF WORDS TO RESTORE
1422 7AFF A   $LP3:   AISZ    R2,-1            ; RELOAD STACK IN REVERSE ORDER
1423 C200 A           LD      R0,(R2)          ; LOAD WORD
1424 6000 A           PUSH    R0               ; PUSH ONTO HARDWARE STACK
1425 79FF A           AISZ    R1,-1            ; CHECK IF FINISHED
1426 19FB T           JMP     $LP3             ; GET NEXT WORD
1427 19E4 T           JMP     $REST            ; RESTORE REGS AND RETURN
              ;
              ;   STORAGE NEEDED
              ;
     1429     $SAV0:  .=.+1                    ; REGISTER 0
     142A     $SAV1:  .=.+1                    ; REGISTER 1
     142B     $SAV2:  .=.+1                    ; REGISTER 2
     142C     $RETA:  .=.+1                    ; RETURN ADDRESS
142C 1438 T   $SPTR:  .WORD   $END+5           ; ADDRESS OF SOFTWARE STACK
     1432     $STAK:  .=.+5                    ; TEMPORARY STORAGE FOR TOP 5 WORDS
1432 142D T   $ADR:   .WORD   $STAK            ; ADDRESS OF TEMR STORAGE
              ;
     0000     $END:   .END

IEN1     0001  A      R0       0000  A      R1       0001  A
R2       0002  A      STFL     0000  A      STKINT   1400  T
$ADR     1432  T      $EMP     1407  T      $END     1433  T
$FULL    1414  T      $LP1     1416  T      $LP2     141C  T
$LP3     1422  T      $REST    140C  T      $RETA    1428  T
$SAV0    1428  T      $SAV1    1429  T      $SAV2    142A  T
$SPTR    142C  T      $STAK    142D  T
NO ERROR LINES
SOURCE CK.= 3940
```

Chapter 3
MICROPROCESSOR INS AND OUTS

PUTTING DATA INTO PACE

The instructions that PACE uses to bring data from memory to its accumulators are also used to bring data from peripherals to its accumulators. Thus, PACE treats alike both memory and peripherals: a LOAD instruction, LD, is executed, which copies data from a specified address into a designated accumulator, as (ACr)←(EA). (See page 2-6.)

A LOAD INDIRECT instruction, LD@, can also be used to transfer data from memory or peripherals into a PACE accumulator, but only into AC0, as (AC0)←((EA)). (See page 2-6.)

TAKING DATA OUT OF PACE

The STORE instruction, ST, is used to transfer data out of the processor, as (EA)←(ACr). (See page 2-6.) Here, the contents of the designated accumulator are transferred to the effective address in either a peripheral or memory.

Again, the STORE INDIRECT instruction, ST@, can be used to transfer data from AC0 to a location in either a peripheral or memory, as ((EA))←(AC0). (See page 2-6.)

CALLING A SUBROUTINE

A subroutine (also called a service routine) is an instruction sequence that performs a specific task, such as, for example, reading characters (or data) from the teletype, then echoing them via print-out on the teletype.

To cause a subroutine to be executed by PACE (or any processor), the program must jump to the address containing the first instruction of the subroutine. The address of the first word of the subroutine is called the "entry point". You can cause the program to move to an entry point by using the JUMP TO SUBROUTINE, JSR, instruction or the JUMP TO SUBROUTINE INDIRECT, JSR@, instruction. (See page 2-5.) The effective address of the jump instruction will specify a subroutine's entry point.

The RETURN FROM SUBROUTINE instruction, RTS, is used primarily to return from subroutines entered by JSR. (See page 2-5.)

Subroutines may also be entered via interrupts, and exited by using the RETURN FROM INTERRUPT instruction, RTI. (See page 2-5.) Getting into and exiting from a subroutine is discussed in the text that follows.

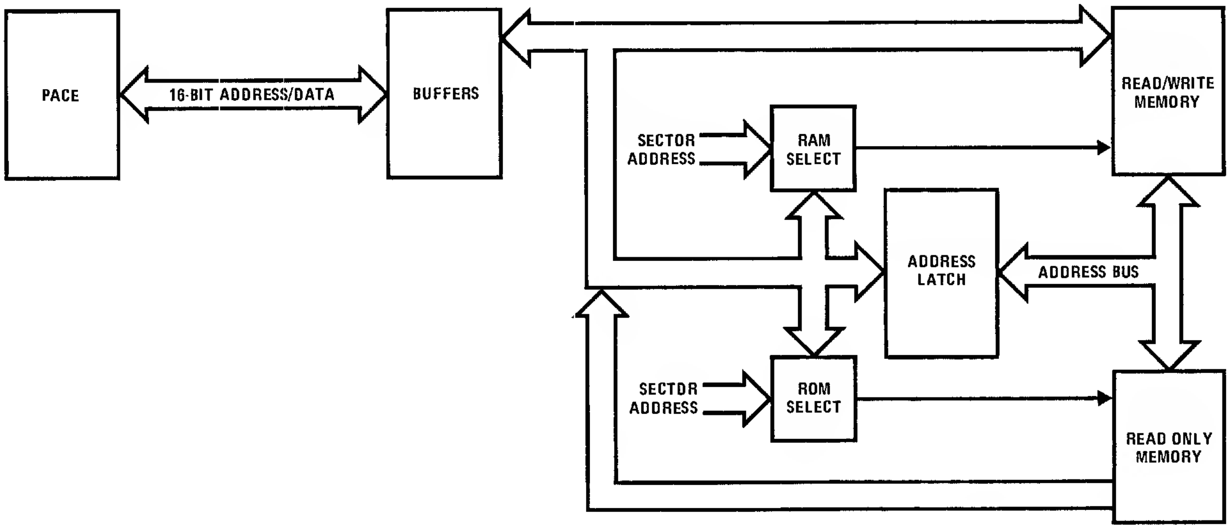


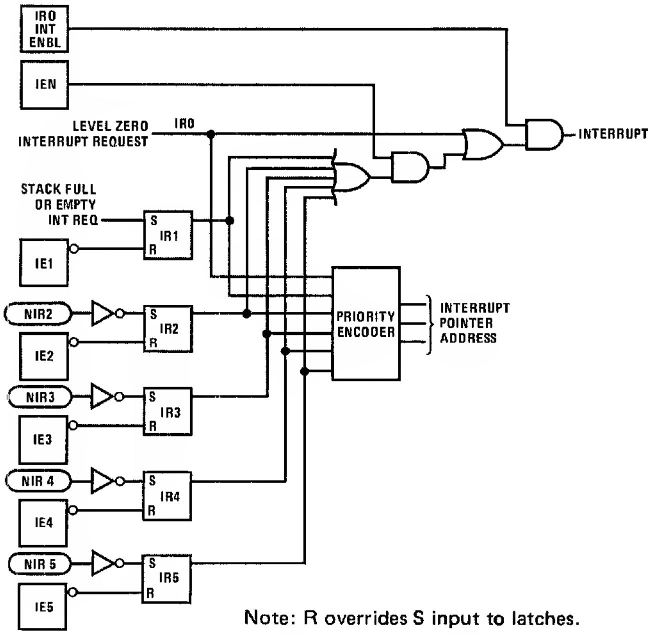
FIGURE 13. PACE Memory Interface

THE INTERRUPT SYSTEM

The PACE microprocessor provides a six-level priority interrupt structure. Each level is provided with an individual Interrupt Enable as shown in *Figure 14.* A master Interrupt Enable (IEN) is provided for all five lower-priority levels at once. Negative true Interrupt Request inputs are provided to allow several interrupts to be wire-ORed to each input. When an Interrupt Request occurs, the associated interrupt request latch (IR1 through IR5) is set if the corresponding Interrupt Enable input is true. Since the interrupt request latch can be set by any pulse exceeding one clock period, narrow timing or control pulses can be captured. If the IEN is true, then an interrupt is generated and recognized after completing the current instruction. During the interrupt sequence, an address is provided by the output from the priority encoder. The address is used to access the interrupt pointer for the highest priority interrupt request (IR0 is highest priority; IR5 is lowest priority). The interrupt pointers are stored in locations 2 through 7 (see Table 13) for Interrupt Requests 1 through 5 and 0, respectively. The interrupt pointer specifies the starting address of the Interrupt Service Routine for the particular interrupt level, except in the case of the Level-0 Interrupt (IR0). (See Chapter 4, PACE Users Manual.) The Level-0 Interrupt is used primarily for Control Panel implementation. Before Interrupt Service Routine execution, the Program Counter contents are pushed onto the stack and IEN is set low (false). This interrupt handling requires 14 microseconds (28 clock cycles). The Interrupt Service Routine may set IEN high (true) after turning off the Interrupt Enable for the interrupt level currently being serviced (or resetting the Interrupt Request). The Interrupt Enable Signals can be set and reset by the Set Flag (SFLG) and Pulse Flag (PFLG) Instructions described on page 2-11. If an Interrupt Enable Flag is set or reset, one more instruction is executed before the interrupt is enabled or disabled. The Return From Interrupt (RTI) instruction may also be used to set IEN true. In this case there is no delay and a pending interrupt will take effect immediately after execution of RTI.

Three types of external interrupts are likely to occur in PACE applications: short-duration (pulse) interrupts; long-duration resettable interrupts; and nonresettable interrupts. The short-duration interrupt exists for less than the interrupt response time and may be caused by a strobe pulse from a peripheral device or the occurrence of a high-speed transient condition, a short-duration interrupt must be latched to be recognized. Interrupts longer than the clock period are latched by the PACE interrupt request latches. The Interrupt Service Routine must reset the interrupt request latch by turning off the Interrupt Enable for the level being serviced. If the Interrupt Enable is left off, Interrupt Request pulses cannot set the interrupt request latch.

Long-duration resettable interrupts last longer than the interrupt response time and may be reset by the Interrupt Service Routine. An example is a Buffer-Full Interrupt by a peripheral device. The Interrupt Service Routine empties the buffer, removing the interrupt. A long-duration interrupt is ignored when Interrupt Enable is low but still generates an interrupt when Interrupt Enable is set true. In servicing long-duration interrupts, the interrupt request latch must be cleared after the interrupt is reset by the Interrupt Service Routine.



Note: R overrides S input to latches.

FIGURE 14. PACE Interrupt System

TABLE 13. Locations of Interrupt Pointers

INTERRUPT POINTER	LOCATION
Interrupt 0 Program	8
Interrupt 0 PC	7
Interrupt 5	6
Interrupt 4	5
Interrupt 3	4
Interrupt 2	3
Interrupt 1	2
Not Assigned	1
Initialization Instruction	0

Long-duration nonresettable interrupts last longer than the interrupt response time and are not reset by the Interrupt Service Routine. An example of a long-duration resettable interrupt is a photoelectric cell that detects the presence of an item on a conveyor. The signal produced by the photoelectric cell (or some other sensor) may last for a significant portion of a second. Setting the interrupt request latch on the edge of the interrupt is desirable and may be accomplished using a simple RC circuit or single-shot to generate a pulse on the edge of the interrupt.

The interrupt response time for PACE is equal to the time to finish the current instruction at the time of the interrupt, plus the time to access the first instruction of the Interrupt Service Routine. Instruction execution times are given on page 2-12.

An example of an Interrupt Service Routine for Interrupt Level 3 is shown in Table 14. Memory location 4 contains the address of the first instruction in the routine. When a Level-3 Interrupt occurs, the first instruction preserves the state of the flags on the stack.

NOTE: IEN is set false by the interrupt prior to being saved on the stack.

The flag data then are loaded into AC0 and all bits which are to be modified are masked out to zero. The desired bits are then set true by ORing with IESTAT. If the routine is interruptable, then IE3 is set to zero and IEN is set to one. The modified status word is then transferred from AC0 to the status register. The actual servicing of the interrupting device then takes place. At the end of the routine, the flags are restored and a return instruction is executed. If the interrupts are to be reenabled, the RTI Instruction must be used since RTI sets IEN true and restores the PC from the stack.

NOTE: Status register masking is necessary only when interrupt enable status is to be modified to allow higher priority devices to interrupt. Pushing the status register onto the stack is necessary only if the routine alters the contents of the status register.

TABLE 14. Interrupt Service Routine Example

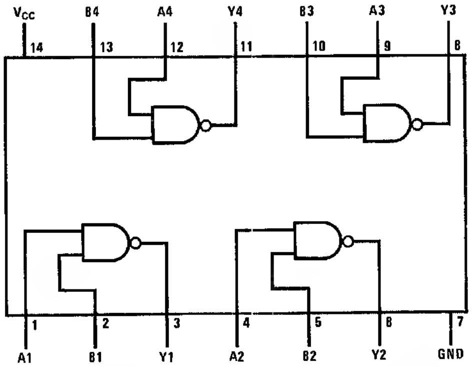
ASSEMBLY CODE			EXPLANATION
	. = 4		Set location counter equal to 4.
	.WORD	ISERV3	Pointer to service routine.
	. = 500		Set location counter equal to 500.
ISERV3:	PUSHF		Save flags on stack.
	CFR	AC0	Move flags to AC0.
	AND	AC0, MASK	Mask out old Interrupt Enable status.
	OR	AC0, IESTAT	OR in new Interrupt Enable status.
	CRF	AC0	Store in flag register.
	.		.
	.		.
	.		.
	.		Interrupt Service Routine
	.		.
	.		.
	.		.
	.		.
	.		.
INTXIT:	PULLF		Restore flags.
	RTI		Return to interrupted routine.
MASK:	.WORD		Mask data
IESTAT:	.WORD		Interrupt Enable Status data

Chapter 4
THE SIMULATIONS

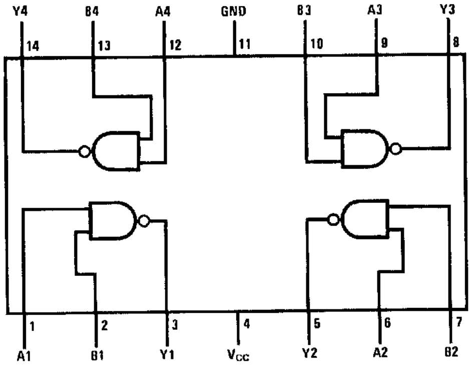
Part 1: STANDARD FUNCTIONS

00 Quad 2-Input NAND Gates

$Y = \overline{AB}$



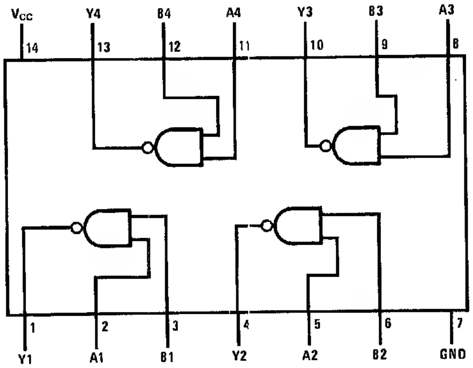
5400/7400(J), (N); 54H00/74H00(J), (N);
54L00/74L00(J), (N); 54LS00/74LS00(J),(N),(W);
74S00(N)



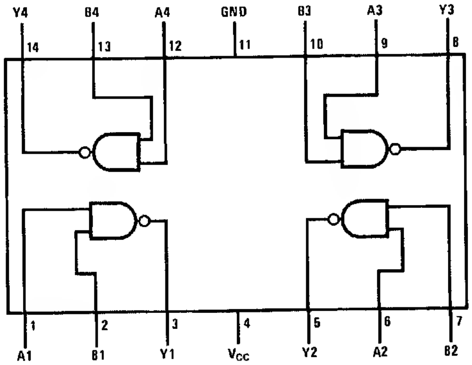
5400/7400(W); 54L00/74L00(W)

01 Quad 2-Input NAND Gates with Open-Collector Outputs

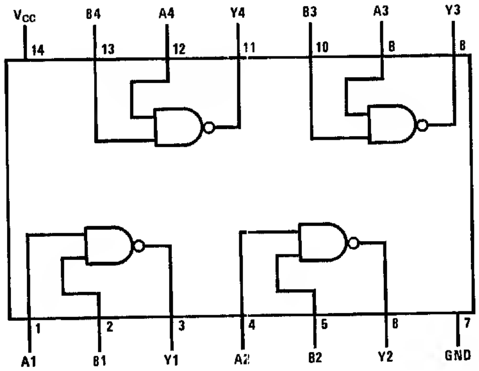
$Y = \overline{AB}$



5401/7401(J), (N); 54LS01/74LS01(J), (N), (W)



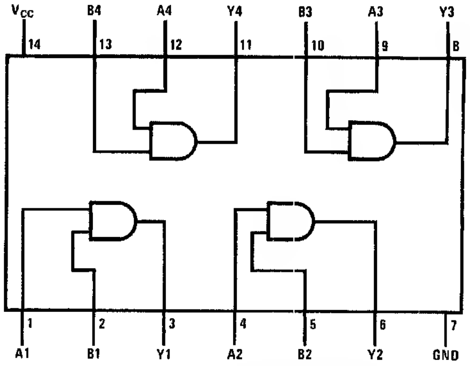
5401/7401(W); 54L01/74L01(W)



54H01/74H01(J), (N)

08 Quad 2-Input AND Gates

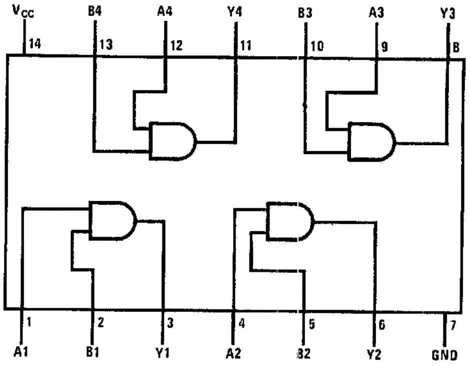
Y = AB



5408/7408(J), (N), (W); 54H08/74H08(J), (N);
54L08/74L08(J), (N), (W);
54LS08/74LS08(J), (N), (W)

09 Quad 2-Input AND Gates with Open-Collector Outputs

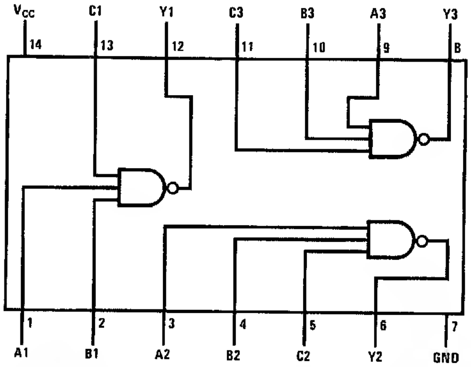
Y = AB



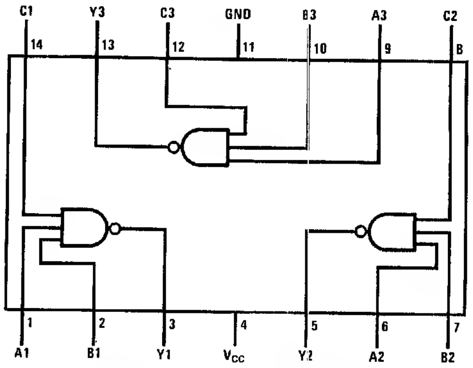
5409/7409(J), (N), (W); 54L09/74L09(J), (N), (W);
54LS09/74LS09(J), (N), (W)

10 Triple 3-Input NAND Gates

Y = $\overline{ABC}$



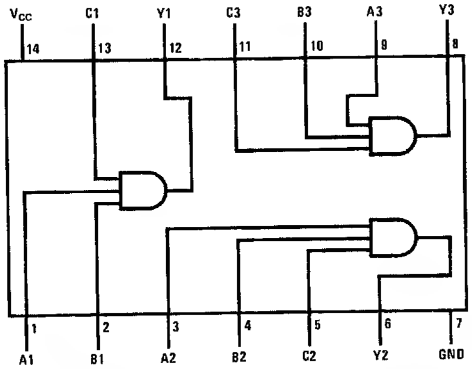
5410/7410(J), (N); 54H10/74H10(J), (N);
54L10/74L10(J), (N); 54LS10/74LS10(J), (N), (W);
74S10(N)



5410/7410(W); 54L10/74L10(W)

11 Triple 3-Input AND Gates

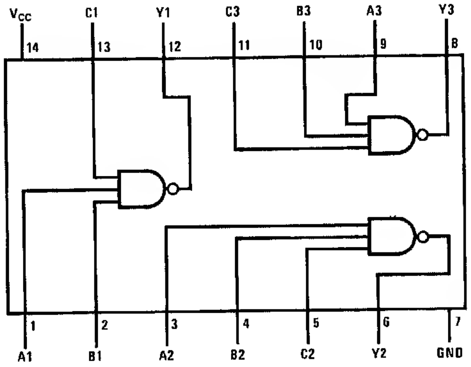
Y = ABC



5411/7411(J), (N); 54H11/74H11(J), (N); 54L11/74L11(J), (N), (W); 54LS11/74LS11(J), (N), (W); 74S11(N)

12 Triple 3-Input NAND Gates with Open-Collector Outputs

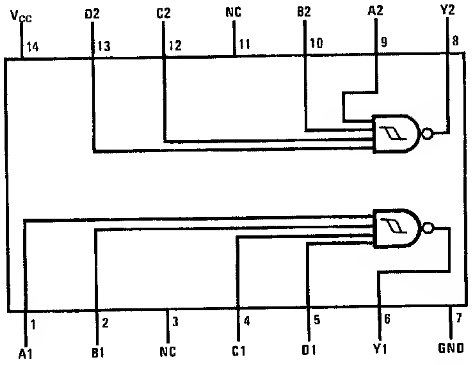
Y = $\overline{ABC}$



54LS12/74LS12(J), (N), (W)

13 Dual 4-Input NAND Schmitt Triggers

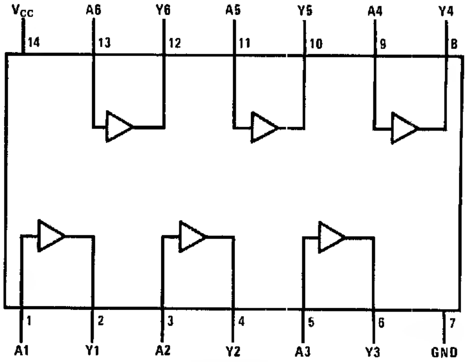
Y = $\overline{ABCD}$



5413/7413(J),(N),(W); 54LS13/74LS13(J),(N),(W)

17 Hex Buffers with Open-Collector High-Voltage Outputs

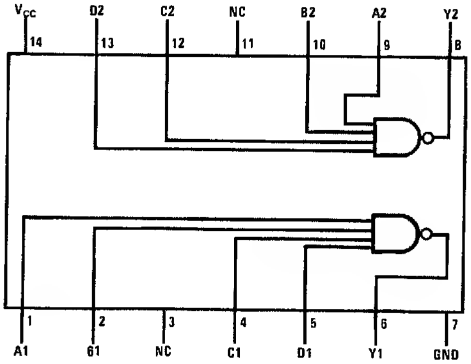
Y = A



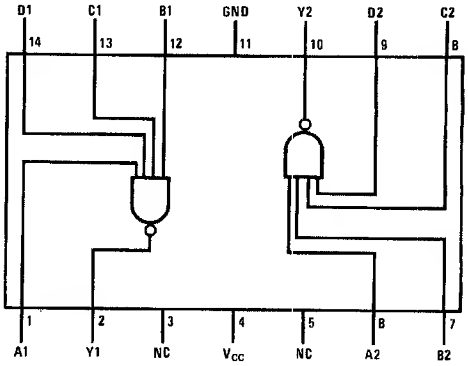
5417/7417(J),(N),(W)

20 Dual 4-Input NAND Gates

Y = $\overline{ABCD}$



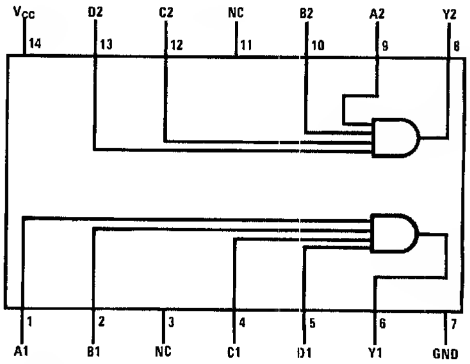
5420/7420(J),(N); 54H20/74H20(J),(N);
54L20/74L20(J),(N); 54LS20/74LS20(J),(N),(W);
74S20(N)



5420/7420(W); 54L20/74L20(W)

21 Dual 4-Input AND Gates

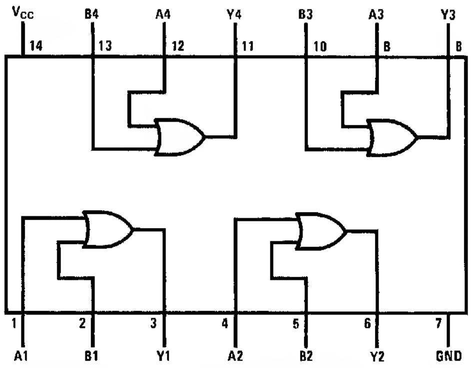
Y = ABCD



54H21/74H21(J),(N);54LS21/74LS21(J),(N),(W)

32 Quad 2-Input OR Gates

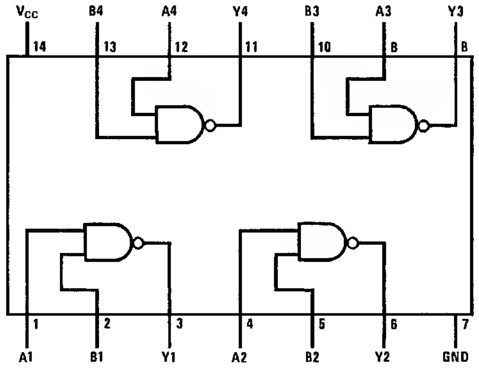
Y = A + B



5432/7432(J),(N),(W);54L32/74L32(J),(N),(W);
54LS32/74LS32(J),(N),(W)

37 Quad 2-Input NAND Buffers

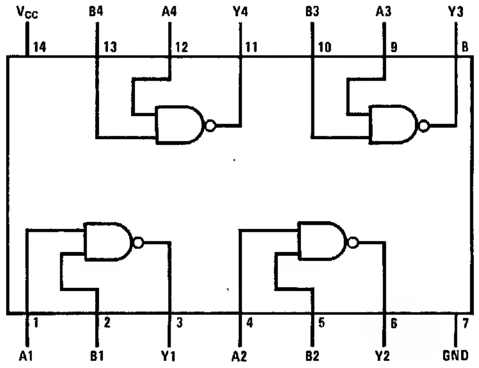
Y = $\overline{AB}$



5437/7437(J),(N),(W);54LS37/74LS37(J),(N),(W)

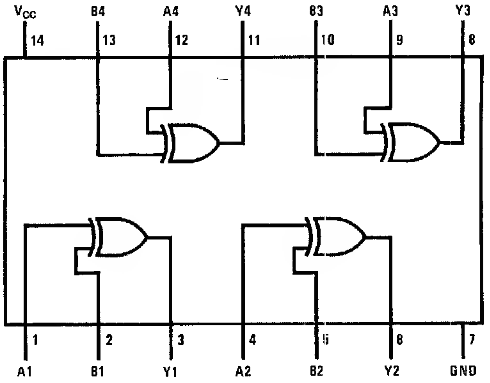
38 Quad 2-Input NAND Buffers with Open-Collector Outputs

Y = $\overline{AB}$

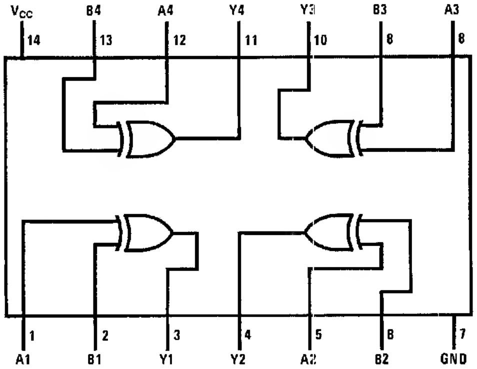


5438/7438(J),(N),(W);54LS38/74LS38(J),(N),(W)

86 Quad 2-Input EXCLUSIVE-OR Gates



5486/7486(J), (N), (W);
54LS86/74LS86(J), (N), (W); 74S86(N)

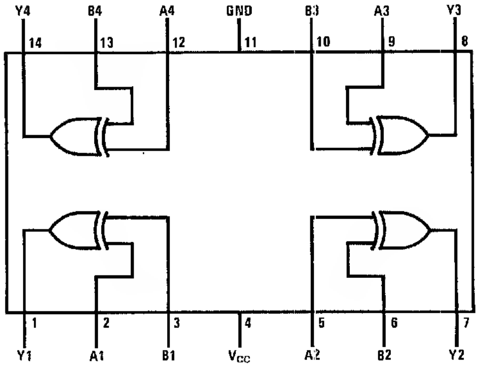


54L86/74L86(J),(N)

TRUTH TABLE
(86, L86, LS86, S86)

INPUTS		OUTPUT
A	B	Y
L	L	L
L	H	H
H	L	H
H	H	L

$Y = A \oplus B = \overline{A}B + A\overline{B}$



54L86/74L86(W)

ACTIVE PULL-UP AND FUNCTION

The DM7408 active pull-up AND function may be implemented by PACE as shown below either by ANDing the contents of two registers or by ANDing the contents of register AC0 with the contents of a memory location.

The contents of two registers may be ANDed by:

RAND, sr, dr ;Contents (A) of source register (sr) are ANDed with contents (B) of destination register (dr). Result (A ∧ B) replaces initial contents (B) of destination register; contents of source register are not altered.

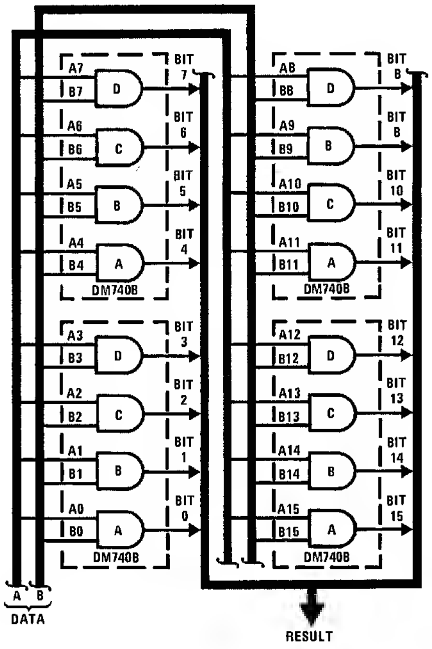
The contents of register AC0 may be ANDed with the contents of a memory location by:

AND 0, disp ;Contents (A) of AC0 are ANDed with contents (B) of memory location specified by displacement (disp) value. Result (A ∧ B) replaces initial contents (A) of AC0; contents of memory location are not altered.

The AND function shown above may be changed to a NAND function by complementing the result as shown below:

CAI r, 00 ;Contents of register (r) are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

PACE Implementation of DM7408 Active Pull-Up AND Function



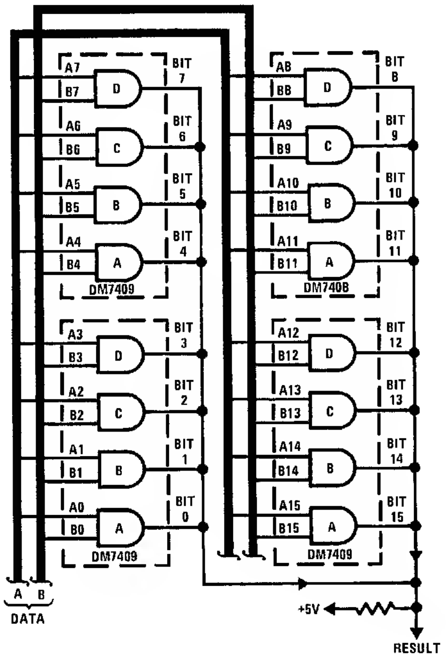
OPEN-COLLECTOR AND FUNCTION

The DM7409 open-collector AND function allows the outputs of several gates to be tied together for input expansion. This function may be implemented by PACE as shown below by ANDing the contents of register AC0 with the contents of a memory location, then complementing the result and testing the complemented result for zero. The contents of register AC0 may be complemented and tested for zero by:

CAI 0, 00 ;Contents of AC0 are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

BOC 1, disp ;Fetch next instruction from memory location specified by displacement (disp) value if contents of AC0 are zero; fetch next instruction in sequence if contents of AC0 are not zero.

PACE Implementation of DM7409 Open-Collector AND Function



Triple 3-Input AND Gate

A 3-input AND function may be implemented by PACE as shown below either by ANDIng the contents of three registers or by ANDing the contents of register AC0 with the contents of two memory locations.

The contents of three registers may be ANDed by:

RAND sr, dr ;Contents (A) of first source register (sr) are ANDed with contents (B) of destination register (dr). Result (A ∧ B) replaces initial contents (B) of destination register; contents of first source register are not altered.

RAND sr, dr ;Contents (C) of second source register are ANDed with contents (A ∧ B) of destination register. Result (A ∧ B ∧ C) replaces initial contents (A ∧ B) of destination register; contents of second source register are not altered.

The contents of AC0 may be ANDed with the contents of two memory locations by:

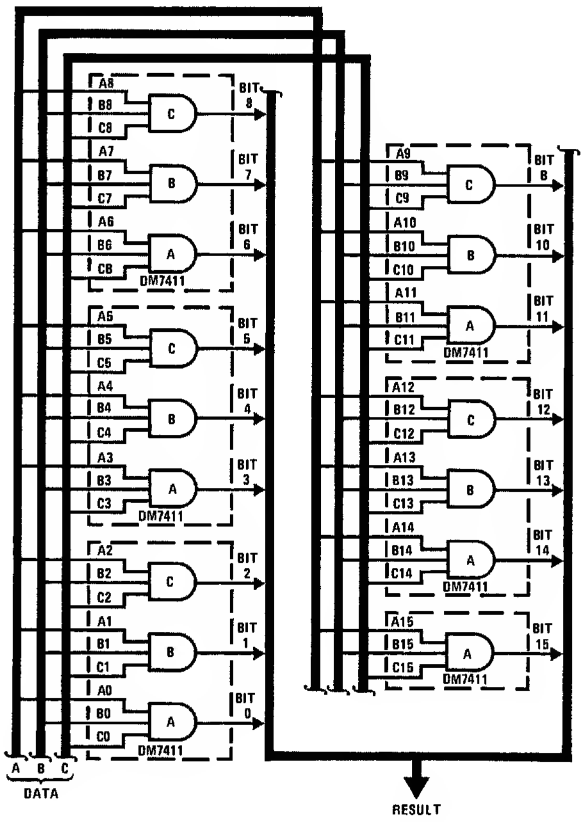
AND 0, disp ;Contents (A) of AC0 are ANDed with contents (B) of memory location specified by displacement (disp) value. Result (A ∧ B) replaces initial contents (A) of AC0; contents of first memory location are not altered.

AND 0, disp ;Contents (A ∧ B) of AC0 are ANDed with contents (C) of memory location specified by displacement value. Result (A ∧ B ∧ C) replaces initial contents (A ∧ B) of AC0; contents of second memory location are not altered.

The AND function shown above may be changed to a NAND function by complementing the result as shown below:

CAI r, 00 ;Contents of register (r) are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

PACE Implementation of 3-Input AND Function



Dual 4-Input AND Gate

A 4-input AND function may be implemented by PACE as shown below either by ANDing the contents of four registers or by ANDing the contents of register AC0 with the contents of three memory locations.

The contents of four registers may be ANDed by:

RAND sr, dr ;Contents (A) of first source register (sr) are ANDed with contents (B) of destination register (dr). Result (A ∧ B) replaces initial contents (B) of destination register; contents of first source register are not altered.

RAND sr, dr ;Contents (C) of second source register are ANDed with contents (A ∧ B) of destination register. Result (A ∧ B ∧ C) replaces initial contents (A ∧ B) of destination register; contents of second source register are not altered.

RAND sr, dr ;Contents (D) of third source register are ANDed with contents (A ∧ B ∧ C) of destination register. Result (A ∧ B ∧ C ∧ D) replaces initial contents (A ∧ B ∧ C) of destination register; contents of third source register are not altered.

The contents of register AC0 may be ANDed with the contents of three memory locations by:

AND 0, disp ;Contents (A) of AC0 are ANDed with contents (B) of memory location specified by displacement (disp) value. Result (A ∧ B) replaces initial contents (A) of AC0; contents of first memory location are not altered.

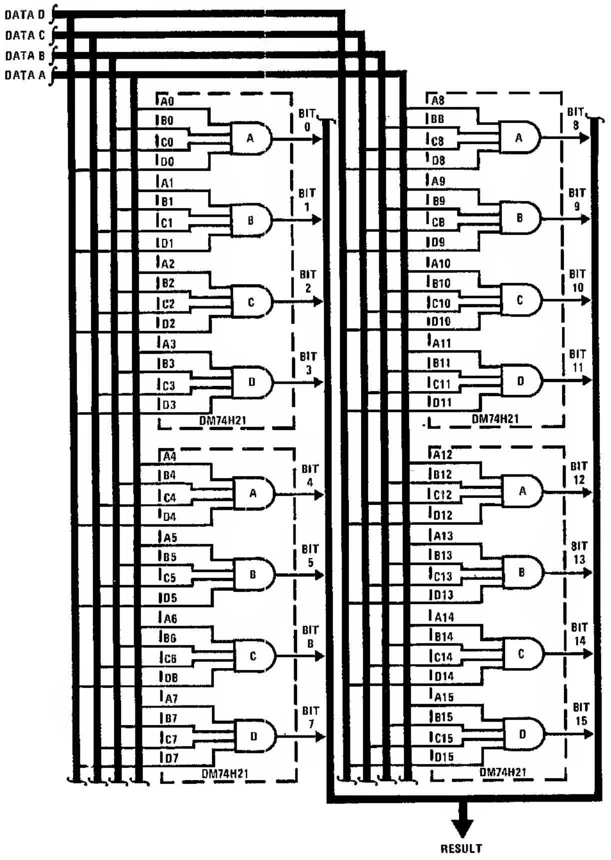
AND 0, disp ;Contents (A ∧ B) of AC0 are ANDed with contents (C) of memory location specified by displacement value. Result (A ∧ B ∧ C) replaces initial contents (A ∧ B) of AC0; contents of second memory location are not altered.

AND 0, disp ;Contents (A ∧ B ∧ C) of AC0 are ANDed with contents (D) of memory location specified by displacement value. Result (A ∧ B ∧ C ∧ D) replaces initial contents (A ∧ B ∧ C) of AC0; contents of third memory location are not altered.

The AND function shown above may be changed to a NAND function by complementing the result as shown below:

CAI r, 00 ;Contents of register (r) are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

PACE Implementation of Four-Input AND Function



A 2-input OR function may be implemented with PACE as shown below by ORing the contents of register AC0 with the contents of a memory location.

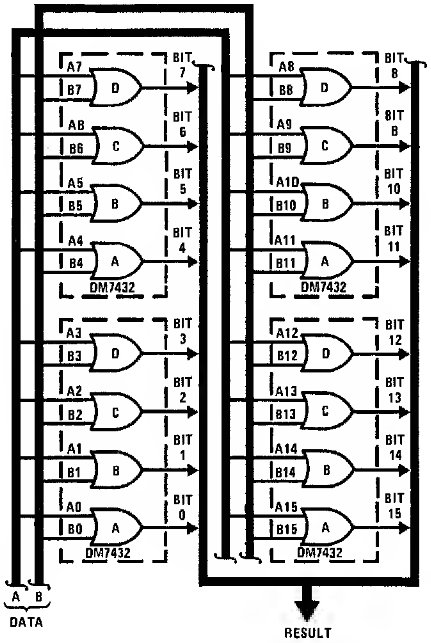
The contents of register AC0 may be ORed with the contents of a memory location by:

OR 0, disp ;Contents (A) of AC0 are ORed with contents (B) of memory location specified by displacement (disp) value. Result (A ∧ B) replaces initial contents (A) of AC0; contents of memory location are not altered.

The OR function shown here may be changed to a NOR function by complementing the result as shown below:

CAI r, 00 ;Contents of register (r) are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

PACE Implementation of 2-Input OR Function



A 2-input EXCLUSIVE-OR function may be implemented by PACE as shown below by exclusively ORing the contents of two registers.

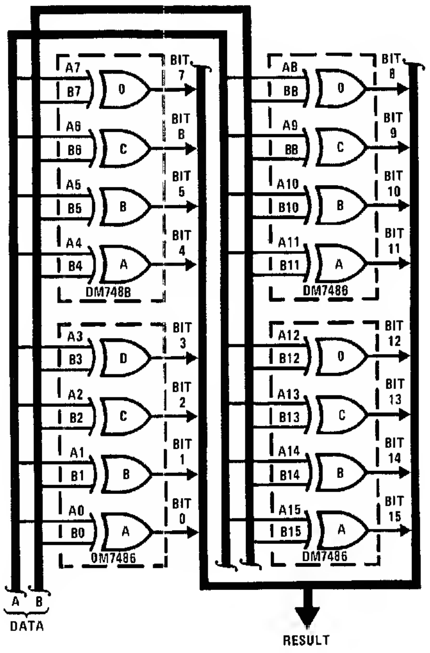
The contents of two registers may be exclusively ORed by:

RXOR sr, dr ;Contents (A) of source register (sr) are exclusively ORed with contents (B) of destination register (dr); result ($A \veebar B$) replaces initial contents (B) of destination register; contents of source register are not altered.

The OR function shown here may be changed to a NOR function by complementing the result as shown below:

CAI r, 00 ;Contents of register (r) are 1's complemented and added to displacement (disp) value zero to maintain 1's complement.

PACE Implementation of 2-Input EXCLUSIVE-OR Function



MSI

DM54/DM7483,LS83A,LS283

4-Bit Binary Adders with Fast Carry

General Description

These full adders perform the addition of two 4-bit binary numbers. The sum (Σ) outputs are provided for each bit and the resultant carry (C4) is obtained from the fourth bit. These adders feature full internal look ahead across all four bits. This provides the system designer with partial look-ahead performance at the economy and reduced package count of a ripple-carry implementation.

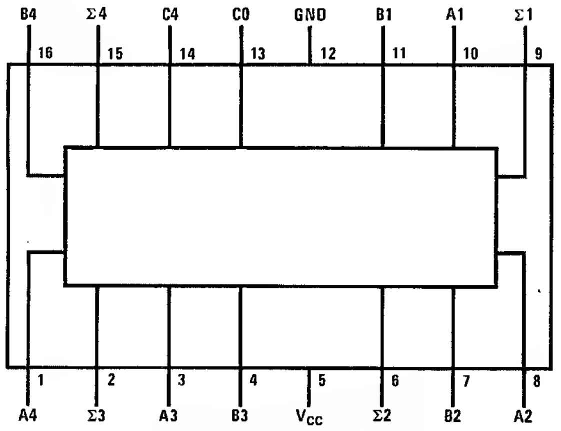
The adder logic, including the carry, is implemented in its true form meaning that the end-around carry can be accomplished without the need for logic or level inversion.

Features

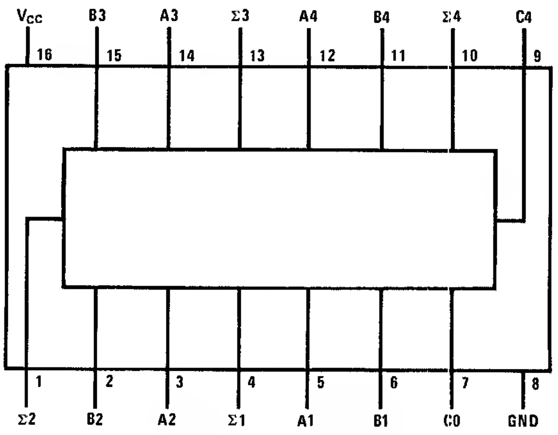
- Full-carry look-ahead across the four bits
- Systems achieve partial look-ahead performance with the economy of ripple carry

TYPE	TYPICAL ADD TIMES TWO 8-BIT WORDS	 TWO 16-BIT WORDS	TYPICAL POWER DISSIPATION PER 4-BIT ADDER
83	23 ns	43 ns	290 mW
LS83A	25 ns	45 ns	95 mW
LS283	25 ns	45 ns	95 mW

Connection Diagrams and Truth Table



5483(J), (W); 7483(J), (N), (W);
54LS83A/74LS83A(J), (N), (W)



54LS283/74LS283(J), (N), (W)

INPUT				OUTPUT					
				WHEN C0 = L / WHEN C2 = L			WHEN C0 = H / WHEN C2 = H		
A1 / A3	B1 / B3	A2 / A4	B2 / B4	Σ1 / Σ3	Σ2 / Σ4	C2 / C4	Σ1 / Σ3	Σ2 / Σ4	C2 / C4
L	L	L	L	L	L	L	H	L	L
H	L	L	L	H	L	L	L	H	L
L	H	L	L	H	L	L	L	H	L
H	H	L	L	L	H	L	H	H	L
L	L	H	L	L	H	L	H	H	L
H	L	H	L	H	H	L	L	L	H
L	H	H	L	H	H	L	L	L	H
H	H	H	L	L	L	H	H	L	H
L	L	L	H	L	H	L	H	H	L
H	L	L	H	H	H	L	L	L	H
L	H	L	H	H	H	L	L	L	H
H	H	L	H	L	L	H	H	L	H
L	L	H	H	L	L	H	H	L	H
H	L	H	H	H	L	H	L	H	H
L	H	H	H	H	L	H	L	H	H
H	H	H	H	L	H	H	H	H	H

H = High Level, L = Low Level

Note : Input conditions at A1, B1, A2, B2, and C0 are used to determine outputs Σ1 and Σ2 and the value of the internal carry C2. The values at C2, A3, B3, A4, and B4 are then used to determine outputs Σ3, Σ4, and C4.

A binary full-adder function (with carry out and overflow) may be implemented with PACE as shown below by adding the contents of two registers (with or without carry in) or by adding the contents of a memory location to the contents of register AC0 (without carry in).

The contents of two registers may be added with carry in by:

RADC sr, dr ;Contents (A) of source register (sr) and carry (CRY) flag are added to contents (B) of destination register (dr). Result (C) replaces initial contents (B) of destination register; contents of source register are not altered. Carry (CRY) and overflow (OV) flags are set or reset according to result.

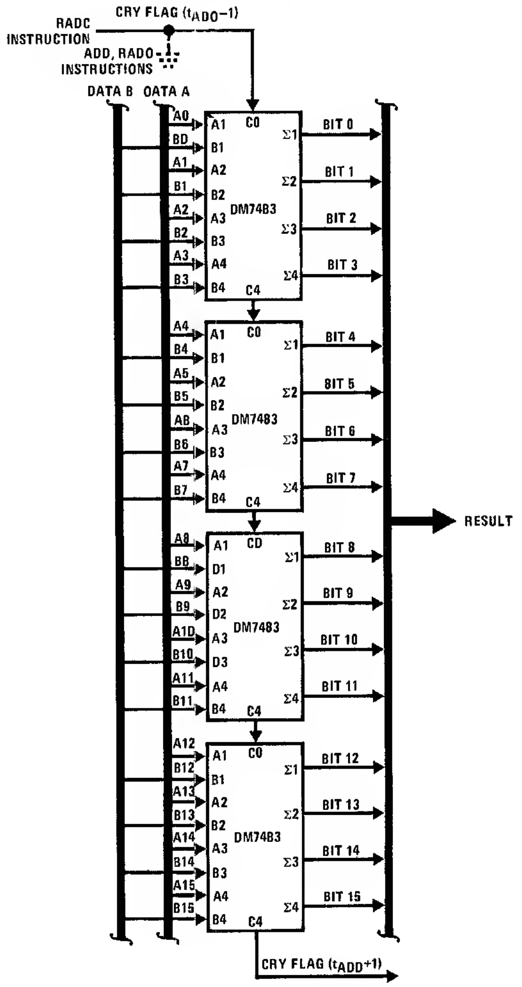
The contents of two registers may be added without carry in by:

RADD sr, dr ;Contents (A) of source register are added to contents (B) of destination register (dr). Result (C) replaces initial contents (B) of destination register; contents of source register are not altered. Carry (CRY) and overflow (OV) flags are set or reset according to result.

The contents of a memory location may be added to the contents of register AC0 by:

ADD 0, disp ;Contents (A) of memory location specified by displacement (disp) value are added to contents (B) of AC0. Result (C) replaces initial contents of AC0; contents of memory location are not altered. Carry (CRY) and overflow (OV) flags are set or reset according to result.

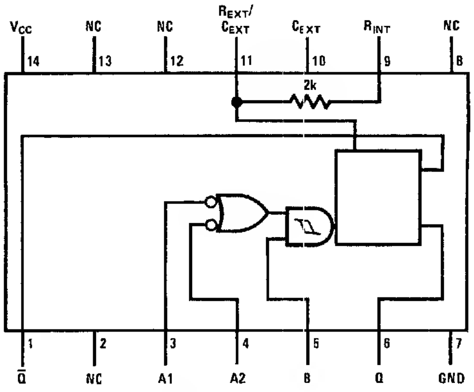
PACE Implementation of Binary Full-Adder Function



121 One Shots

TRUTH TABLE

INPUTS			OUTPUTS	
A1	A2	B	Q	$\overline{Q}$
L	X	H	L	H
X	L	H	L	H
X	X	L	L	H
H	H	X	L	H
H	↓	H	⎍	⊔
↓	H	H	⎍	⊔
↓	↓	H	⎍	⊔
L	X	↑	⎍	⊔
X	L	↑	⎍	⊔

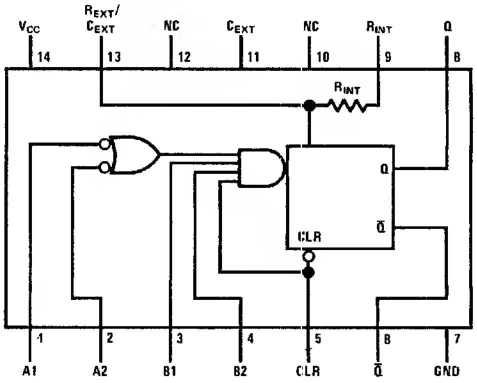


54121/74121(J), (N), (W)

122 Retriggerable One Shots with Clear

TRUTH TABLE

INPUTS					OUTPUTS	
CLEAR	A1	A2	B1	B2	Q	$\overline{Q}$
L	X	X	X	X	L	H
X	H	H	X	X	L	H
X	X	X	L	X	L	H
X	X	X	X	L	L	H
X	L	X	H	H	L	H
H	L	X	↑	H	⎍	⊔
H	L	X	H	↑	⎍	⊔
H	X	L	H	H	L	H
H	X	L	↑	H	⎍	⊔
H	X	L	H	↑	⎍	⊔
H	H	↓	H	H	⎍	⊔
H	↓	↓	H	H	⎍	⊔
H	↓	H	H	H	⎍	⊔
↑	L	X	H	H	⎍	⊔
↑	X	L	H	H	⎍	⊔

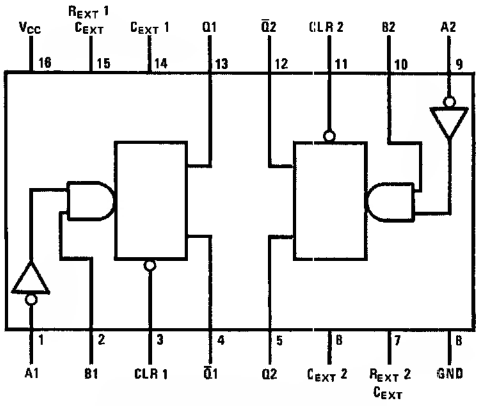


54LS122(J), (W); 74LS122(J), (N)

123,123A Dual Retriggerable One Shots with Clear

TRUTH TABLE

INPUTS			OUTPUTS	
A	B	CLR	Q	$\overline{Q}$
H	X	H	L	H
X	L	H	L	H
L	↑	H	⎍	⊔
↓	H	H	⎍	⊔
X	X	L	L	H



54123/74123(J), (N),, (W);
54L123A/74L123A(J), (N), (W);
54LS123/74LS123(J), (N), (W)

Notes: ⎍ = one high-level pulse, ⊔ = one low-level pulse.
To use the internel timing resistor of 54121/74121, connect R_{INT} to V_{CC}.
An external timing capacitor may be connected between C_{EXT} and R_{EXT}/C_{EXT} (positive).
For accurate repeatable pulse widths, connect an external resistor between R_{EXT}/C_{EXT} and V_{CC} with R_{INT} open-circuited.
To obtain variable pulse widths, connect external verieble resistence between R_{INT} or R_{EXT}/C_{EXT} and V_{CC}.

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The DM74121 is a gated monostable multivibrator capable of providing a jitter-free output pulse ranging from 30 ns to 40 seconds in duration. Selection of the desired pulse width is accomplished by connection of an external RC network with:

$$t_p \text{ (OUT)} = C_T R_T \log_e 2.$$

NOTE: The timing values specified in the descriptions that follow are based on a clock period of 500 ns and an input-output, data-transfer Extend Cycle of 500 ns. For clock periods and/or Extend Cycles of different duration, new timing values can be calculated from the instruction execution time formulas provided in Table 7 (pp. 2-12).

ASSIGNMENTS

The monostable multivibrator function may be implemented with PACE as two separate subroutines that allow selection of a delay interval ranging from 1 ms to approximately 18 hours (in 1-ms increments). The flowchart and program listing that follow assume that accumulator AC0 is used as an input-data register and as a working register (for entry of the desired delay in seconds or milliseconds, and derivation of the corresponding delay loop constant, respectively), and that input/output assignments are as listed below.

Delay in 1-second increments

DM74121	PACE
Trigger	SECOND entry to Delay subroutine
Pulse Width	Execution time of Delay subroutine as selected by decimal value of AC0 contents (for example, a 60-second delay is selected by loading 60_{10} into AC0)

Delay in 1 ms increments

DM74121	PACE
Trigger	MILLISECOND entry to Delay subroutine
Pulse Width	Execution time of Delay subroutine as selected by decimal value of AC0 contents (for example, a 60 ms delay is selected by loading 60_{10} into AC0)

Delays of less than 1 ms can be achieved by inserting Jump + 1 (jump to next instruction in sequence) and/or Shift instructions directly into the main program. Execution time for the Jump (JMP) + 1 instruction is 8.5μs; execution time for a Shift Left (SHL) or Shift Right (SHR) instruction varies according to the number of shifts performed. A shift of 0 is, in effect, a do-nothing instruction that is executed in 12.5μs. For shifts of 1 to 127 places, execution time is computed from the following formula:

$$10.5 + 6n\ \mu s, \text{ where } n = \text{number of shifts performed}$$

Thus, a single Jump + 1 instruction can be used to select the minimum delay of 8.5μs, a single shift instruction can be used to select a delay of 12.5μs or a delay interval ranging from 16.5μs to 772.5μs (in 6.0μs increments), and a combination of Shift and/or Jump + 1 instructions can be used to fine tune the delay interval over the range of 8.5μs to 1 ms.

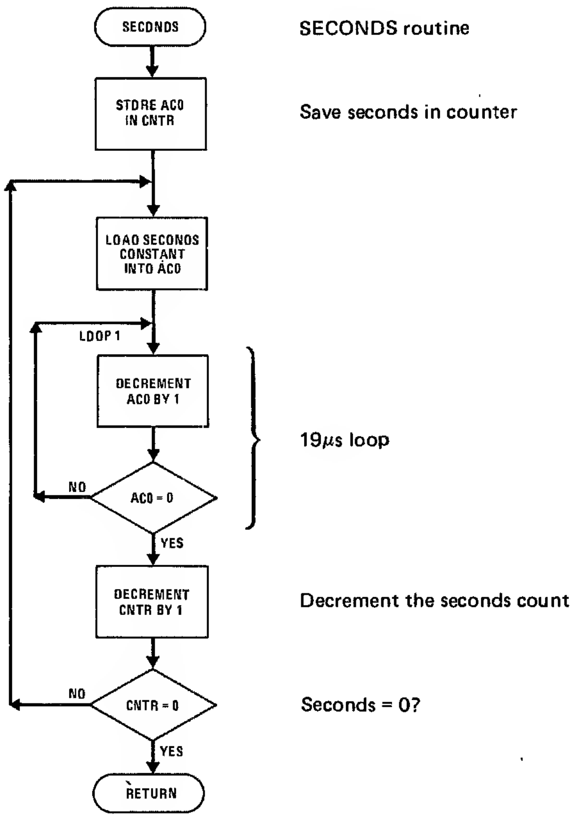
FUNCTIONAL OPERATION

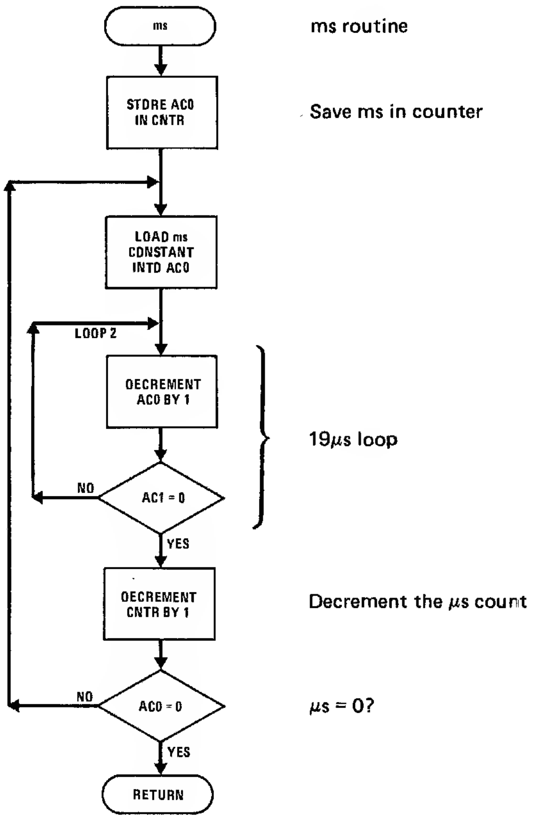
This program is written as two separate subroutines that select a delay interval ranging from 1 ms to approximately 1 minute (in 1 ms increments), or from 1 second to approximately 18 hours (in 1 second increments). When either subroutine is called by the main program, it is assumed that the desired delay interval has already been loaded into AC0. The first instruction executed for either subroutine, therefore, saves the contents of AC0 in memory-location CNTR to free AC0 for use as a working register. AC0 is then loaded with the value 51_{10} (MSECS subroutine) or $52{,}630_{10}$ (SECS subroutine), and decremented by one at a 19μs rate to provide either a 1 ms or 1 second delay cycle. When the contents of AC0 equal zero, the delay value stored in CNTR is decremented by one and the delay cycle/decrement CNTR sequence is repeated until the contents of CNTR equal zero.

Decrementing of AC0 at a 19μs rate is accomplished via an AISZ −1 instruction followed by a JMP, Loop 1 instruction. While AC0 is being decremented to zero, execution times for the AISZ and JMP instructions are 10.5μs and 8.5μs, respectively. Upon detection of AC0 = zero, AISZ instruction-execution time increases to 12.5μs to provide an automatic skip to the instruction following the JMP instruction. Thus a DSZ instruction (15.5 or 17.5μs for CNTR > or = 0, respectively) is executed to decrement the contents of CNTR by one. If the new value in CNTR is not zero, the JMP instruction (8.5μs execution time) following the DSZ instruction causes the subroutine to loop back to the MSECS + 1 or SECS + 1 address, thereby enabling another delay cycle/decrement counter sequence. When the contents of CNTR are subsequently decremented to zero, the JMP instruction that follows the DSZ instruction is skipped and an RTS instruction is executed to cause a return to the main program.

The 1 ms and 1 second delay cycles mentioned above are approximations that yield a worst case accuracy of 1% or better over the complete range of delay intervals that can be selected via the subroutine. If greater than 1% accuracy is required for system applications, the subroutine can be used to establish a time base that is slightly less than the desired delay interval, then a combination of Jump and/or Shift instructions can be inserted in the main program to fine tune the delay interval to the desired final value.

FLOW CHARTS





PROGRAM LISTING

```
                  ;        MONOSTABLE MULTIVIBRATOR
         0000     ACØ      =       Ø
                  ;        SECONDS
 0000 D10E A      SECS:    ST      ACØ,CNTR         ;SAVE ACØ IN CNTR
 0001 C105 A               LD      ACØ,D52630       ;LOAD SECOND CONSTANT
 0002 78FF A      LOOP1:   AISZ    ACØ,-1           ;19 MICROSEC LOOP
 0003 19FE A               JMP     LOOP1            ;
 0004 AD0A A               DSZ     CNTR             ;NUMBER OF SECS = Ø
 0005 19FB A               JMP     SECS+1           ;NO, CONTINUE
 0006 8000 A               RTS                      ;YES, RETURN
 0007 CD96 A      D52630:  .WORD   52630            ;DECIMAL 52630
                           ;MILLISECONDS
 0008 D106 A      MSECS:   ST      ACØ,CNTR         ;SAVE ACØ IN CNTR
 0009 5033 A               LI      ACØ,51           ;LOAD MILLISEC CONSTANT
 000A 78FF A      LOOP2:   AISZ    ACØ,-1           ;19 MICROSEC LOOP
 000B 19FE A               JMP     LOOP2            ;
 000C AD02 A               DSZ     CNTR             ;NUMBER OF MILLISECS = Ø
 000D 19FB A               JMP     MSECS+1          ;NO, CONTINUE
 000E 8000 A               RTS                      ;YES, RETURN
 000F 0000 A      CNTR:    .WORD   Ø                ;DELAY COUNTER SAVE WORD
         0000              .END
```

DM54/DM74150,151A,LS151,S151

Data Selectors/Multiplexers

General Description

These data selectors/multiplexers contain full on-chip decoding to select the desired data source. The 150 selects one-of-sixteen data sources; the 151A, LS151, and S151 select one-of-eight data sources. The 150, 151A, LS151, and S151 have a strobe input which must be at a low logic level to enable these devices. A high level at the strobe forces the W output high, and the Y output (as applicable) low.

The 151A, LS151, and S151 feature complementary W and Y outputs whereas the 150 has an inverted (W) output only.

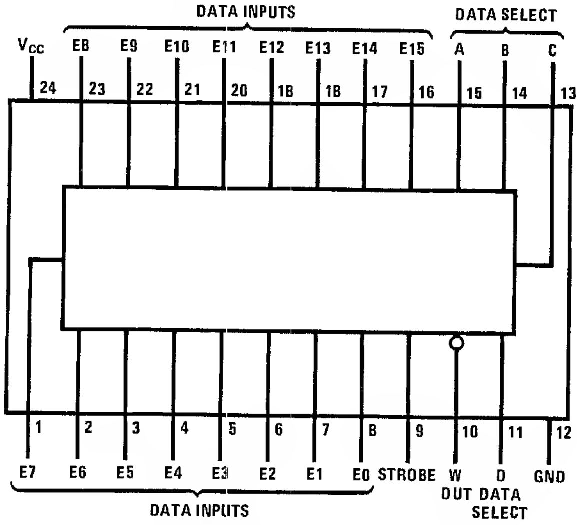
The 151A incorporates address buffers which have symmetrical propagation delay times through the complementary paths. This reduces the possibility of transients occurring at the output(s) due to changes made at the select inputs, even when the 151A outputs are enabled (i.e., strobe low).

Features

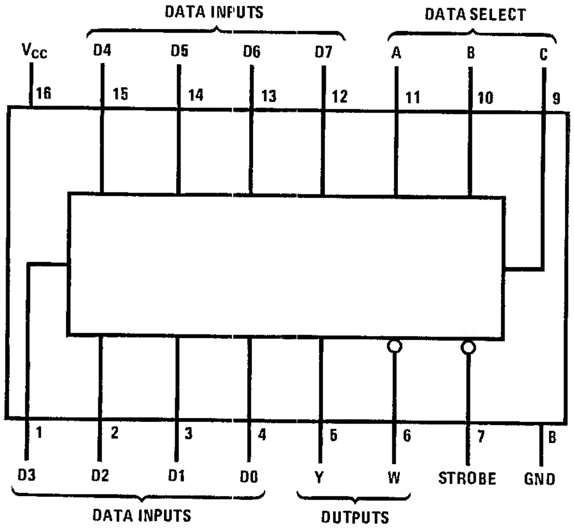
- 150 selects one-of-sixteen data lines
- Others select one-of-eight data lines
- Performs parallel-to-serial conversion
- Permits multiplexing from N lines to one line
- Also for use as Boolean function generator

TYPE	TYPICAL AVERAGE PROPAGATION DELAY TIME DATA INPUT TO W OUTPUT	TYPICAL POWER DISSIPATION
150	11 ns	200 mW
151A	9 ns	135 mW
LS151	12.5 ns	30 mW
S151	4.5 ns	225 mW

Connection Diagrams



54150(J), (F); 74150(J), (N), (F)



54151A(J), (W); 74151A(J), (N), (W);
54LS151/74LS151(J), (N), (W); 74S151(N)

Truth Tables

54150/74150

INPUTS					OUTPUT
SELECT				STROBE	W
D	C	B	A	S	
X	X	X	X	H	H
L	L	L	L	L	$\overline{E0}$
L	L	L	H	L	$\overline{E1}$
L	L	H	L	L	$\overline{E2}$
L	L	H	H	L	$\overline{E3}$
L	H	L	L	L	$\overline{E4}$
L	H	L	H	L	$\overline{E5}$
L	H	H	L	L	$\overline{E6}$
L	H	H	H	L	$\overline{E7}$
H	L	L	L	L	$\overline{E8}$
H	L	L	H	L	$\overline{E9}$
H	L	H	L	L	$\overline{E10}$
H	L	H	H	L	$\overline{E11}$
H	H	L	L	L	$\overline{E12}$
H	H	L	H	L	$\overline{E13}$
H	H	H	L	L	$\overline{E14}$
H	H	H	H	L	$\overline{E15}$

54151A/74151A, 54LS151/74LS151, 74S151

INPUTS				OUTPUTS	
SELECT			STROBE	Y	W
C	B	A	S		
X	X	X	H	L	H
L	L	L	L	D0	$\overline{D0}$
L	L	H	L	D1	$\overline{D1}$
L	H	L	L	D2	$\overline{D2}$
L	H	H	L	D3	$\overline{D3}$
H	L	L	L	D4	$\overline{D4}$
H	L	H	L	D5	$\overline{D5}$
H	H	L	L	D6	$\overline{D6}$
H	H	H	L	D7	$\overline{D7}$

H = High Level, L = Low Level, X = Don't Care
$\overline{E0}$, $\overline{E1}$ . . . $\overline{E15}$ = the complement of the level of the respective E input
D0, D1 . . . D7 = the level of the respective D input

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The DM74150 functions under control of the STROBE input to provide 16-line to 1-line data multiplexing. While the STROBE is low, the four Data Select inputs (A, B, C, D) are continuously decoded to route the appropriate data input (E0 through E15) to the output (W); when the STROBE is high, decoding is disabled and the output is held in the high state.

ASSIGNMENTS

The DM74150 multiplexer function may be implemented with PACE using AC0 as an input data register and AC1 as an input/output data register. The flowchart and program listing that follow assume that the AC0 and AC1 bit positions are assigned as listed below:

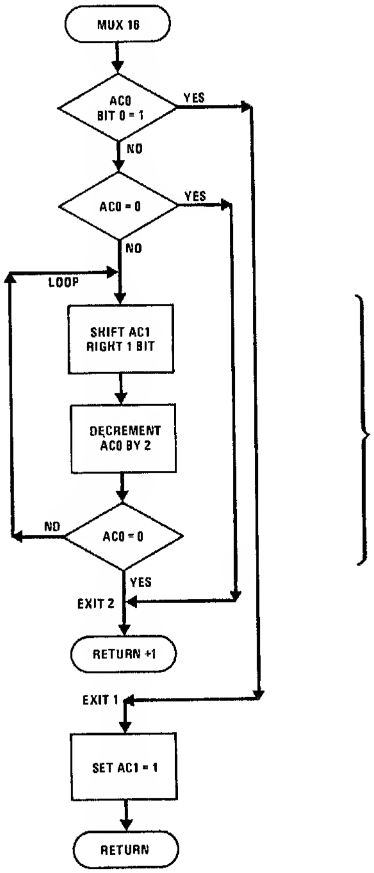
INPUTS:		OUTPUT:	
DM74150	PACE	DM74150	PACE
STROBE	AC0 Bit 0	W	AC1 Bit 0
A	AC0 Bit 1		
B	AC0 Bit 2		
C	AC0 Bit 3		
D	AC0 Bit 4		
E0	AC1 Bit 0		
•	•		
•	•		
•	•		
E15	AC1 Bit 15		

FUNCTIONAL OPERATION

This program is written as a subroutine that performs 16-line to 1-line multiplexing. It is assumed that when the subroutine is called, the main operating program has already loaded the STROBE and Data Select inputs into AC0 and the Data inputs into AC1. The first step of the subroutine is to test AC0 Bit 0 via a Branch-On-Condition (BOC) instruction. If AC0 Bit 0 is high, AC1 Bit 0 is set high to reflect logical operation of the DM74150 in response to a high STROBE input, and a Return From Subroutine (RTS) instruction is executed to provide a "Multiplexing Disabled" return to the main operating program.

If AC0 Bit 0 is low, AC1 is rotated right while AC0 is decremented by two after each shift until the contents of AC0 are equal to zero. A decrement of two is required because the AC0 STROBE bit is low and the least significant Data Select Bit (A) is located at AC0 bit position 1. This bit position corresponds to a binary arithmetic value of 2^2. Thus, when the contents of AC0 are equal to zero, the selected data input will have been rotated to bit position 0 of AC1. Upon detection of AC0 = 0, a Return From Subroutine (RTS) + 1 instruction is executed to provide a "Multiplexed Data Valid" return to the main operating program.

FLOW CHART



Strobe high?

Selected data bit is located at bit position 0 of AC1

Shift AC1 right until selected data bit is located at bit position 0

Multiplexed Data Valid return

Set output high

Multiplexing Disabled return

PROGRAM LISTING

```
                  ;         16 TO 1 MULTIPLEXER
        ØØØØ      ACØ       =        Ø
        ØØØ1      AC1       =        1
  ØØØØ  43Ø5  A   MUX16:    BOC      3,EXIT1        ;EXIT IF STROBE = 1
  ØØØ1  41Ø3  A             BOC      1,EXIT2        ;EXIT IF ACØ = Ø
  ØØØ2  2DØ2  A   LOOP:     SHR      AC1,1,Ø        ;SHIFT AC1 RIGHT 1 BIT
  ØØØ3  78FE  A             AISZ     ACØ,-2         ;DECREMENT ACØ BY 2
  ØØØ4  19FD  A             JMP      LOOP           ;CONTINUE TESTING
  ØØØ5  8ØØ1  A   EXIT2:    RTS      1              ;MUX RETURN
  ØØØ6  51Ø1  A   EXIT1:    LI       AC1,1          ;SET OUTPUT = NO MUX
  ØØØ7  8ØØØ  A             RTS                     ;NO MUX RETURN
        ØØØØ                .END
```

MSI

DM54/DM74154,L154A,LS154

4-Line to 16-Line Decoders/Demultiplexers

General Description

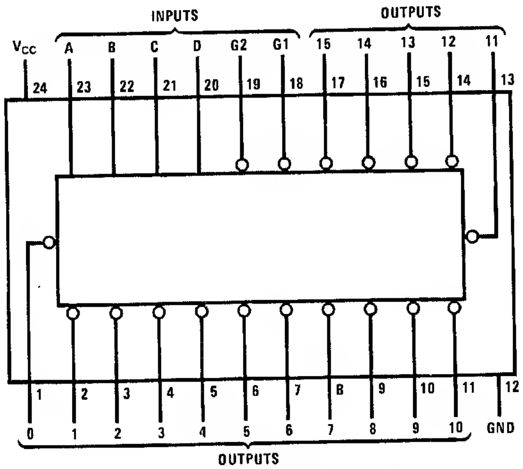
Each of these 4-line-to-16-line decoders utilizes TTL circuitry to decode four binary-coded inputs into one of sixteen mutually exclusive outputs when both the strobe inputs, G1 and G2, are low. The demultiplexing function is performed by using the 4 input lines to address the output line, passing data from one of the strobe inputs with the other strobe input low. When either strobe input is high, all outputs are high. These demultiplexers are ideally suited for implementing high-performance memory decoders. All inputs are buffered and input clamping diodes are provided to minimize transmission-line effects and thereby simplify system design.

Features

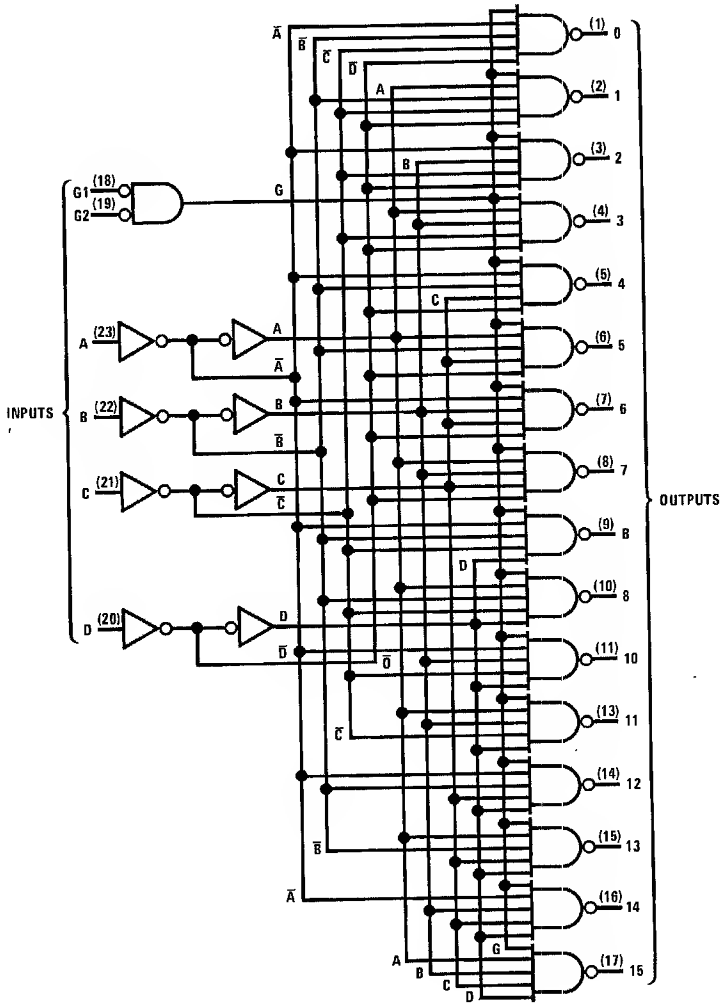
- Decodes 4 binary-coded inputs into one of 16 mutually exclusive outputs
- Performs the demultiplexing function by distributing data from one input line to any one of 16 outputs
- Input clamping diodes simplify system design
- High fan-out, low-impedance, totem-pole outputs

TYPE	TYPICAL PROPAGATION DELAY 3 LEVELS OF LOGIC	TYPICAL PROPAGATION DELAY STROBE	TYPICAL POWER DISSIPATION
154	19 ns	18 ns	170 mW
L154A	55 ns	45 ns	24 mW
LS154	23 ns	19 ns	45 mW

Connection and Logic Diagrams



54154(J), (F); 74154(J), (N), (F);
54L154A/74L154A(J), (N), (F);
54LS154/74LS154(J), (N), (F)



HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The DM74154 has six inputs and sixteen outputs. Two of the inputs, G1 and G2, serve as enable inputs. When both of these inputs are low, the remaining four inputs (A, B, C, and D) are decoded to provide a low level (logic 0) at the appropriate output pin.

ASSIGNMENTS

The DM74154 decoder/demultiplexer function may be implemented with PACE, using AC0 as an input/output data register and AC1 as a working data register. The PACE decoder/demultiplexer flowchart and program listing that follow assume that the AC0 bit positions are assigned as listed below:

INPUTS:		OUTPUTS:	
	PACE		PACE
DM74154	AC0 Bit	DM74154	AC0 Bit
G1	0	0	0
G2	1	1	1
A	2	2	2
B	3	.	.
C	4	.	.
D	5	.	.
		15	15

FUNCTIONAL OPERATION

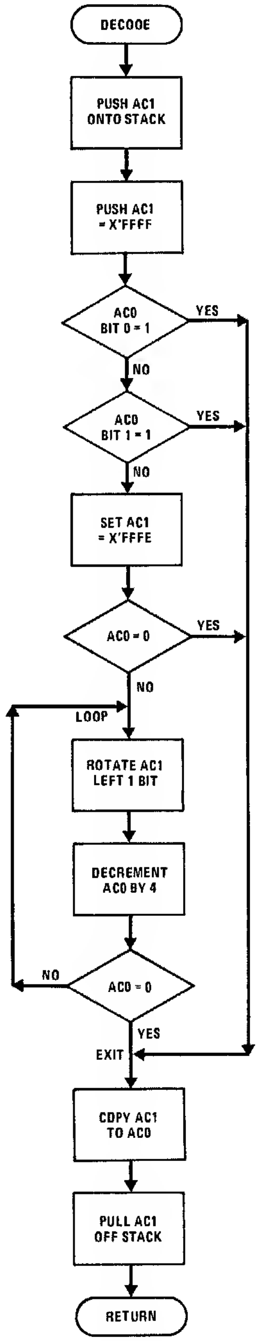
This program is written as a subroutine that performs 4-line to 16-line decoding. It is assumed that when the subroutine is called, the main program has already loaded the decode enable and data inputs into AC0 according to the assignment specified previously. Since the subroutine requires that AC1 be used as a working register, the first operation of the subroutine is to push AC1 onto the stack so that the original contents of AC1 can be restored at the end of the subroutine.

After the original contents of AC1 are stored on the stack, all sixteen bits of AC1 are set high and the AC0 G1 and G2 bits are tested for the zero (low) state. If either bit is high, no decoding occurs and AC1 is copied into AC0 to set all sixteen bits of AC0 high. Then AC1 is pulled from the stack to restore the original contents and the subroutine is exited with AC0 set to FFFF to indicate that an invalid decode was detected.

If both the AC0 G1 and G2 bits are low, bit 0 of AC1 is set low to initiate the decode sequence, then the contents of AC0 are tested for zero to determine whether bit 0 is the selected output. If the contents of AC0 are zero, AC1 is copied into AC0 to complete the decode sequence and the subroutine is exited after the original contents of AC1 are restored from the stack. If the contents of AC0 are not zero, further decoding is accomplished by rotating AC1 left while decrementing AC0 by four after each shift until the contents of AC0 equal zero. A decrement of four is required because the AC0 G1 and G2 bits are zero and the least significant AC0 data select bit (A) is located at bit position 2, which, in effect, multiplies the value of the A-D data select bits by a factor of four. Thus, a decrement of four cancels the multiplication factor without the use of additional instructions and a zero value in AC0 indicates that the low-level bit in AC1 has been rotated to the appropriate output position.

Upon detection of AC0 = 0, the contents of AC1 are copied into AC0, AC1 is pulled from the stack to restore the original contents, and the subroutine is exited with the results of the decode stored in AC0.

FLOW CHART



Save contents of AC1

Set all bits of AC1 high

G1 = 1?

G2 = 1?

Set bit 0 of AC1 low

Test AC0 A-through-D bits for 0

Rotate AC1 low-level bit to output position that corresponds to value of AC0 A-through-D bits

Transfer output to AC0

Restore AC1

PROGRAM LISTING

```
                      ;        4 TO 16 DECODE/DEMULTIPLEX
          ØØØØ        ACØ      =       Ø
          ØØØ1        AC1      =       1
   ØØØØ   61ØØ  A     DECODE:  PUSH    AC1            ;SAVE AC1 ON STACK
   ØØØ1   51FF  A              LI      AC1,ØFF        ;SET AC1 = FFFF
   ØØØ2   43Ø6  A              BOC     3,EXIT         ;BRANCH IF G1 = 1
   ØØØ3   44Ø5  A              BOC     4,EXIT         ;BRANCH IF G2 = 1
   ØØØ4   51FE  A              LI      AC1,ØFE        ;SET AC1 = FFFE
   ØØØ5   41Ø3  A              BOC     1,EXIT         ;EXIT IF ACØ = Ø
   ØØØ6   21Ø2  A     LOOP:    ROL     AC1,1,Ø        ;ROTATE AC1 LEFT 1
   ØØØ7   78FC  A              AISZ    ACØ,-4         ;DECREMENT ACØ BY 4
   ØØØ8   19FD  A              JMP     LOOP           ;CONTINUE TESTING
   ØØØ9   5C4Ø  A     EXIT:    RCPY    AC1,ACØ        ;SAVE RESULTS IN ACØ
   ØØØA   65ØØ  A              PULL    AC1            ;RESTORE AC1 FROM STACK
   ØØØB   8ØØØ  A              RTS                    ;RETURN
          ØØØØ                 .END
```

9-Bit Parity Generators/Checkers

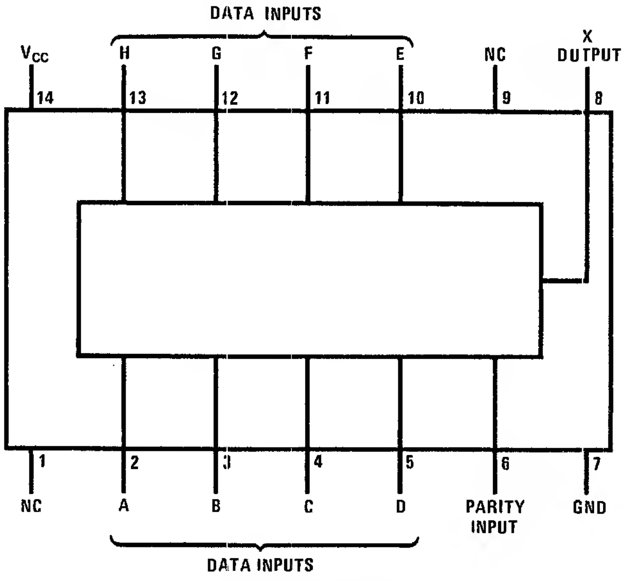
General Description

These circuits can be used both to check for parity and to generate a parity bit. When the generation of a parity bit is desired, the eight data inputs are connected to the transmission lines. If a low logic level is then connected to the parity input, the circuit will generate odd parity. The succeeding parity checker will acknowledge an odd number of "1's" (odd parity) with a low logic level on its output. If a high logic level is connected to the parity input of the first parity generator, the parity checker will acknowledge even parity with a high logic level on its output, although the output of the parity generator will be low.

Features

- Typical propagation delay 34 ns
- Typical power dissipation 130 mW

Connection Diagram



7220/8220(J), (N), (W)

Truth Table

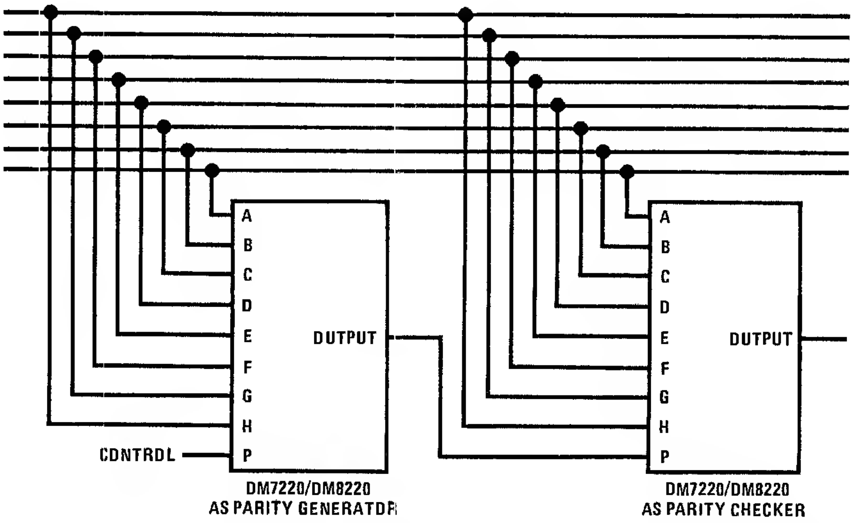
PARITY INPUT	OUTPUT*	INPUTS A THRU H
H	L	Even number of inputs are High
L	L	Odd number of inputs are High

*Single device

Typical Application

If the control line is a logical "0" the parity generator will generate odd parity. The parity checker will acknowledge the presence of an odd number of "1's" (odd parity) with a logical "0" on its output.

If the control line is a logical "1" the parity generator will generate even parity. The parity checker will acknowledge the presence of an even number of "1's" (even parity) with a logical "1" on its output.



HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The DM8220 can be used either to generate parity or to check parity. As shown in the truth table below, it continually processes the PARITY INPUT along with the 8-bit Data Input to provide a high OUTPUT in response to an even number of ones, and a low OUTPUT in response to an odd number of ones. Thus, when the DM8220 is used as a parity generator, the PARITY INPUT is preset to the high or low state to select even or odd parity, respectively; when the DM8220 is used as a parity checker, even parity is indicated by a high output and odd parity is indicated by a low output.

PARITY INPUT	8-BIT DATA INPUT	OUTPUT	PARITY GENERATED	PARITY DETECTED
High	Odd "1"s	High	Even	Even
High	Even "1"s	Low	Even	Odd
Low	Odd "1"s	Low	Odd	Odd
Low	Even "1"s	High	Odd	Even

ASSIGNMENTS

The DM8220 parity generation/detection function may be implemented with PACE using accumulator AC0 as an input/output data register and accumulators AC1 and AC2 as working registers. The flowchart and program listing that follow assume that the AC0 bit positions are assigned as follows:

Parity Generation

INPUTS:		OUTPUT:	
DM8220	PACE	DM8220	PACE
INPUT A	AC0 Bit 0	OUTPUT	AC0 Bit 8
•	•		
•	•		
•	•		
INPUT H	AC0 Bit 7		
PARITY INPUT	Program word TYPE		

Parity Detection

INPUTS:		OUTPUT:	
DM8220	PACE	DM8220	PACE
INPUT A	AC0 Bit 0	OUTPUT	Parity check = RTS + 1
•	•		Parity error = RTS
•	•		
•	•		
INPUT H	AC0 Bit 7		
PARITY INPUT	AC0 Bit 8		

FUNCTIONAL OPERATION

This program is written as a subroutine that either generates or checks parity. Both functions require that the type of parity desired (even or odd) be set previously by the subroutine SETPTY. The following examples show the use of SETPTY:

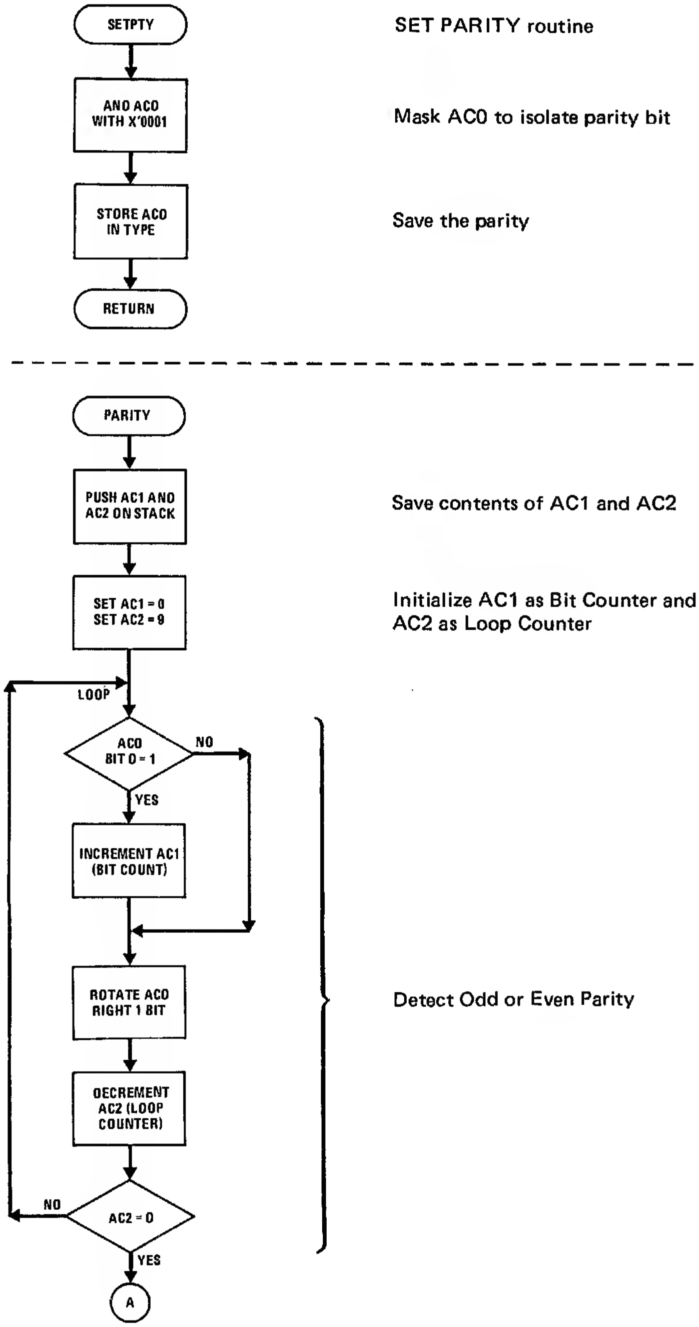
```
LI AC0, 0        ;Load odd parity into AC0
JSR SETPTY       ;Set parity
   or
LI AC0, 1        ;Load even parity into AC0
JSR SETPTY       ;Set parity
```

Since the subroutine PARITY can be used both to generate and detect parity, functional implementation of the subroutine requires that the programmer take into account the types of outputs provided. For parity generation purposes, bit 8 of AC0 serves as a parity output since it is always set to reflect the type of parity selected (e.g., if even parity is selected and AC0 bits 0 through 8 equal an odd number of logic ones, AC0 bit 8 is set high during execution of the subroutine; if even parity is selected and AC0 bits 0 through 8 equal an even number of logic ones, the logical state of AC0 bit 8 is not changed during execution of the subroutine.) For parity detection purposes, bit 8 of AC0 serves as the ninth bit of the input data word and the RTS and RTS + 1 exits from the subroutine serve to indicate, respectively, whether a parity error or valid parity was detected. The examples that follow the program listing indicate how the outputs of the subroutine are typically processed for parity generation and for parity detection.

When the subroutine PARITY is called by the main program, it is assumed that the input data word has already been loaded into AC0. The first step of the subroutine, therefore, is to push working registers AC1 and AC2 onto the stack so that the original contents of AC1 and AC2 can be restored at the end of the subroutine. After AC1 and AC2 are pushed on the stack, AC1 is initialized to zero for use as a bit counter and AC2 is initialized to nine for use as a loop counter. AC0 is then rotated right while AC2 is decremented to zero to allow the logic state of each input bit to be tested and AC1 to be incremented each time that a logic-one bit is detected. Thus, when AC2 = 0, bit 0 of AC1 will be high if an odd number of logic-one bits were detected and low if an even number of logic-one bits were detected. Upon detection of AC2 = 0, AC1 is shifted right one place with link to preserve the status of bit 0 in the link. Then AC1 is pulled from the stack to restore its original contents, AC0 is rotated right to return the data word to the assigned location, and the contents of AC2 (zero) are compared with the contents of memory location TYPE via a Skip If Not Equal (SKNE) instruction to determine whether even or odd parity is required for the main program. Depending on the type of parity required, the link bit is tested either for the high or low state to allow valid parity/parity error detection.

When even parity is required, a low state for the link bit indicates valid parity and a high state indicates a parity error; when odd parity is required, the opposite is true. Thus, if the state of the link bit indicates valid parity, AC2 is pulled from the stack to restore the original contents and the subroutine is exited via a Return From Subroutine (RTS) + 1 instruction to provide a valid parity return to the main program. If the state of the link bit indicates a parity error, bit 8 of AC2 is set high and the contents of AC2 are Exclusively OR'ed with the contents of AC0 to change the state of output parity bit 8. Then AC2 is pulled from the stack to restore the original contents, and the subroutine is exited via a Return From Subroutine (RTS) instruction to provide a parity error return to the main program.

FLOW CHART



SET PARITY routine

Mask AC0 to isolate parity bit

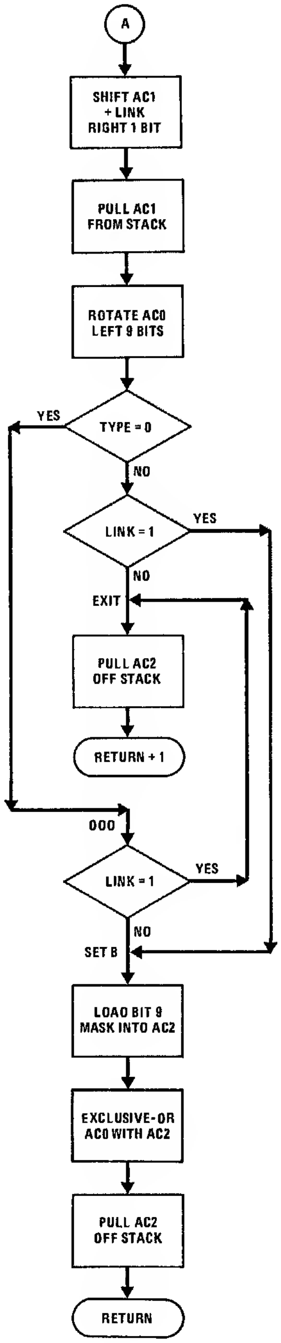
Save the parity

Save contents of AC1 and AC2

Initialize AC1 as Bit Counter and AC2 as Loop Counter

Detect Odd or Even Parity

FLOW CHART (Continued)



Set link high or low to indicate odd or even parity, respectively

Restore original contents of AC1

Rotate data input to original position

Check type of parity selected

Even parity selected

Is count odd?

Restore original contents of AC2

Odd parity selected

Is count odd?

Toggle Bit 8 in AC0

Restore original contents of AC2

Error Return if parity check

PROGRAM LISTING

```
                  ;          PARITY  CHECKER/GENERATOR
       ØØØØ       ACØ        =        Ø
       ØØØ1       AC1        =        1
       ØØØ2       AC2        =        2
       ØØØ8       LINK       =        8
ØØØØ   62ØØ  A    PARITY:    PUSH     AC2             ;SAVE REGISTERS ON STACK
ØØØ1   61ØØ  A               PUSH     AC1             ;
ØØØ2   51ØØ  A               LI       AC1,Ø           ;SET BIT COUNT = Ø
ØØØ3   43Ø1  A    LOOP:      BOC      3,LP1           ;BRANCH IF ACØ BIT Ø = 1
ØØØ4   19Ø1  A               JMP      LP2             ;
ØØØ5   79Ø1  A    LP1:       AISZ     AC1,1           ;INCREMENT BIT COUNTER
ØØØ6   24Ø2  A    LP2:       ROR      Ø,1,Ø           ;ROTATE ACØ RIGHT 1 BIT
ØØØ7   7AFF  A               AISZ     AC2,-1          ;DECREMENT LOOP COUNTER
ØØØ8   19FA  A               JMP      LOOP            ;AC2 NOT ZERO
ØØØ9   2DØ3  A               SHR      AC1,1,1         ;PUT LSB OF AC1 IN LINK
ØØØA   65ØØ  A               PULL     AC1             ;RESTORE AC1 FROM STACK
ØØØB   2Ø12  A               ROL      ACØ,9,Ø         ;REPOSITION INPUT DATA
ØØØC   F9ØE  A               SKNE     AC2,TYPE        ;SKIP IF PARITY IS EVEN
ØØØD   19Ø3  A               JMP      ODD             ;PARITY IS ODD
                  ;          EVEN PARITY
ØØØE   48Ø3  A               BOC      LINK,SET8       ;IF COUNT ODD, SET EVEN
ØØØF   66ØØ  A    EXIT:      PULL     AC2             ;RESTORE AC2 FROM STACK
ØØ1Ø   8ØØ1  A               RTS      1               ;NORMAL RETURN
                  ;          ODD PARITY
ØØ11   48FD  A    ODD:       BOC      LINK,EXIT       ;IF COUNT ODD, RETURN
ØØ12   C9Ø3  A    SET8:      LD       AC2,$Ø1ØØ       ;LOAD MASK INTO AC2
ØØ13   588Ø  A               RXOR     AC2,ACØ         ;TOGGLE ACØ BIT 8
ØØ14   66ØØ  A               PULL     AC2             ;RESTORE AC2 FROM STACK
ØØ15   8ØØØ  A               RTS                      ;CHECK ERROR RETURN
ØØ16   Ø1ØØ  A    $Ø1ØØ:     .WORD    Ø1ØØ            ;BIT 8 MASK
                  ;          SET PARITY ROUTINE
ØØ17   A9Ø2  A    SETPTY:    AND      ACØ,$ØØØ1       ;ZERO BITS 15 THRU 1
ØØ18   D1Ø2  A               ST       ACØ,TYPE        ;SAVE PARITY IN TYPE
ØØ19   8ØØØ  A               RTS                      ;RETURN
ØØ1A   ØØØ1  A    $ØØØ1:     .WORD    1               ;MASK
ØØ1B   ØØØØ  A    TYPE:      .WORD    Ø               ;PARITY TYPE SAVE
       ØØØØ                  .END
```

MSI

DM54/DM7485,L85,LS85

4-Bit Magnitude Comparators

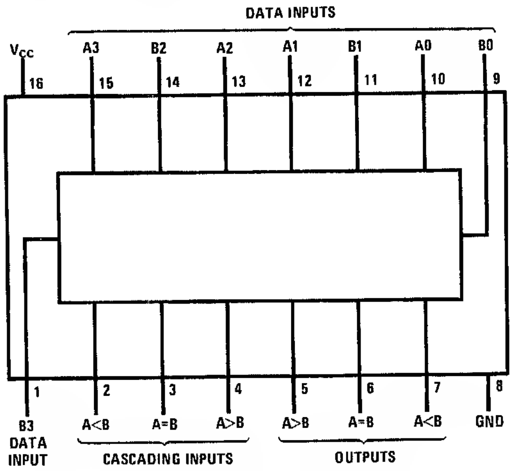
General Description

These four-bit magnitude comparators perform comparison of straight binary or BCD codes. Three fully-decoded decisions about two, 4-bit words (A, B) are made and are externally available at three outputs. These devices are fully expandable to any number of bits without external gates. Words of greater length may be compared by connecting comparators in cascade. The A > B, A < B, and A = B outputs of a stage handling less-significant bits are connected to the corresponding inputs of the next stage handling more-significant bits. The stage handling the least-significant bits must have a high-level voltage applied to the A = B input and in addition for the L85, low-level voltages applied to the A > B and A < B inputs. The cascading paths of the 85, and LS85 are implemented with only a two-gate-level delay to reduce overall comparison times for long words.

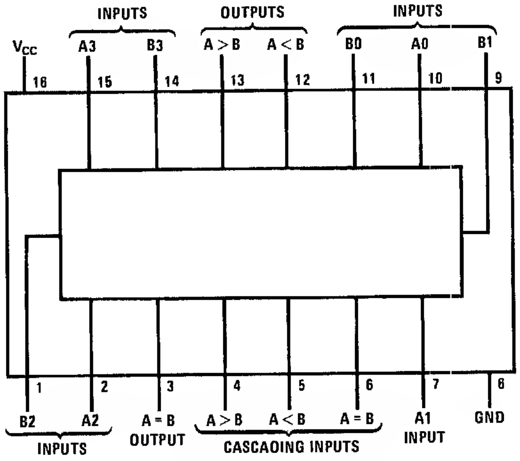
Features

TYPE	TYPICAL POWER DISSIPATION	TYPICAL DELAY (4-BIT WORDS)
85	275 mW	23 ns
L85	20 mW	55 ns
LS85	52 mW	24 ns

Connection Diagrams



5485(J), (W); 7485(J), (N), (W);
54LS85/74LS85(J), (N), (W)



54L85/74L85(J), (N), (W)

Truth Tables

COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A3, B3	A2, B2	A1, B1	A0, B0	A > B	A < B	A = B	A > B	A < B	A = B
A3 > B3	X	X	X	X	X	X	H	L	L
A3 < B3	X	X	X	X	X	X	L	H	L
A3 = B3	A2 > B2	X	X	X	X	X	H	L	L
A3 = B3	A2 < B2	X	X	X	X	X	L	H	L
A3 = B2	A2 = B2	A1 > B1	X	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 > B0	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 < B0	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	L	L	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	H	L	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	H	L	L	H

85, LS85

A3 = B3	A2 = B2	A1 = B1	A0 = B0	X	X	H	L	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	L	L	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	L	H	H	L

L85

A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	H	H	L	H	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	L	H	H	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	H	H	H	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	L	H	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	L	L	L	L

H = High Level, L = Low Level, X = Don't Care

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows four DM7485s cascaded to form a 16-bit magnitude comparator. For this configuration, each 7485 individually compares the four input-variable-A bits with the four-input-variable-B bits. If the two inputs are not equal, the A > B OUT or the A < B OUT line is set high to reflect the appropriate condition. If the two inputs are equal, the A > B OUT, the A = B OUT, or the A < B OUT line is set high according to which of the corresponding inputs is high. For the low-order 7485, the input configuration shown enables the A = B IN line to dominate when equality exists. Thus, the high output from the high-order 7485 reflects the results of the total 16-bit comparison.

ASSIGNMENTS

The 16-bit magnitude comparison function may be implemented with PACE using AC1 as an input data register, AC0 as an input/output data register and AC2 as a working register. The flowchart and program listing that follow assume that the AC0 and AC1 bit positions are assigned as listed below.

INPUTS

	DM7485	PACE		DM7485	PACE
	Input Variable A	AC0 Bit		Input Variable B	AC1 Bit
(LSB)	Bit 0	0	(LSB)	Bit 0	0
	Bit 1	1		Bit 1	1
	Bit 2	2		Bit 2	2
	8it 3	3		Bit 3	3
	Bit 4	4		8it 4	4
	Bit 5	5		Bit 5	5
	8it 6	6		Bit 6	6
	Bit 7	7		Bit 7	7
	8it 8	8		Bit 8	8
	Bit 9	9		Bit 9	9
	Bit 10	10		Bit 10	10
	Bit 11	11		Bit 11	11
	Bit 12	12		Bit 12	12
	Bit 13	13		Bit 13	13
	Bit 14	14		Bit 14	14
(MS8)	8it 15	15		Bit 15	15

OUTPUTS

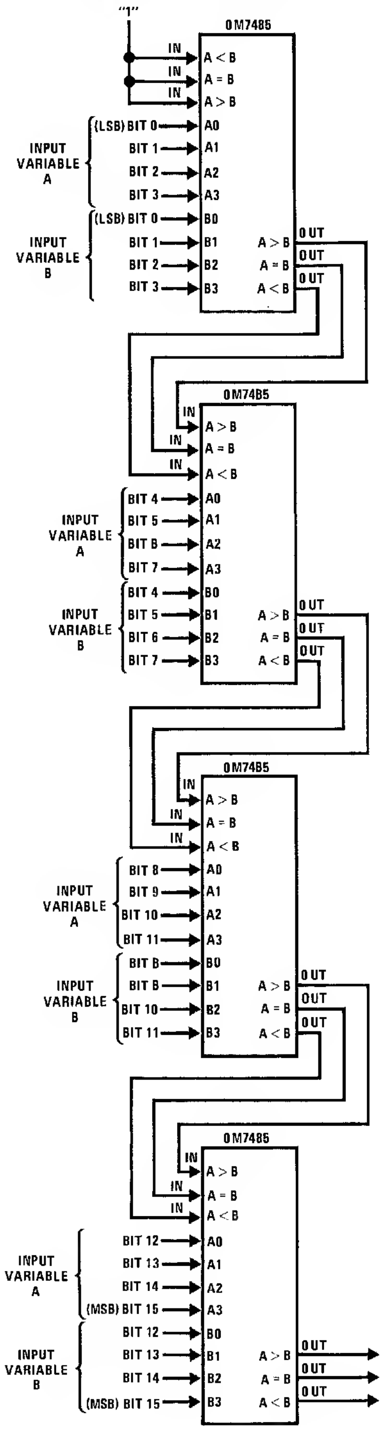
DM7485	· PACE
	AC0 Bit
A > B	1 = High
A = B	0 = High
A < 8	2 = High

FUNCTIONAL OPERATION

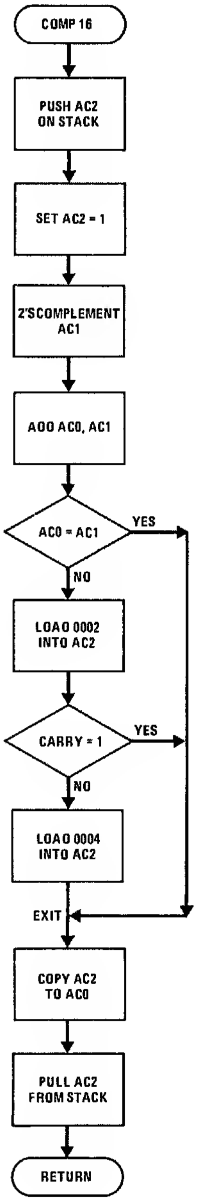
This program is written as a subroutine that compares the magnitude of two 16-bit numbers. It is assumed that when the subroutine is called, the main program has already loaded the A and B values to be compared into accumulators AC0 and AC1, respectively. Since the subroutine requires that AC2 be used as a working register, the first operation of the subroutine is to push AC2 onto the stack so that the original contents of AC2 can be restored at the end of the subroutine. Bit 1 of AC2 is then set to 1, AC1 is subtracted from AC0 using the Complement (CAI) and Register Add (RADD) instructions, and the results are stored in AC0. Use of the CAI and RADD instructions allows the contents of the accumulators to be treated as unsigned numbers to the extent that the carry flag is set whenever the absolute binary value of AC0 is greater than that of AC1.

After the subtraction is performed, AC0 is tested for zero to see if the original A and B values were equal. If AC0 = 0, AC2 is copied into AC0 to set bit 0 of AC0 high, thereby indicating that A = B. If AC0 ≠ 0, bit 1 of AC2 is set high and the carry flag is tested to determine whether A > B or A < B. If the carry flag is set, A > B, and AC2 is copied into AC0 to set bit 1 of AC0 high. If the carry flag is reset, A < B, so bit 2 of AC2 is set high before AC2 is copied into AC0. After being copied into AC0, AC2 is pulled from the stack to restore the original contents, and the subroutine is exited with the results of the comparison stored in AC0.

DM7485 Interconnection for 16-Bit Magnitude Comparison



FLOW CHART



Save contents of AC2

AC2 Bit 0 set high

AC1 subtracted from AC0

A = B?

AC2 Bit 1 set high

AC0 > AC1 (carry set)?

AC2 Bit 2 set high

Results of comparison stored in AC0

Restore AC2 from stack

PROGRAM LISTING

```
                         ;        16 BIT COMPARATOR
            0000         AC0      =      0
            0001         AC1      =      1
            0002         AC2      =      2
     0000   6200   A     COMP16:  PUSH   AC2           ;SAVE AC2 ON STACK
     0001   5201   A              LI     AC2,1         ;SET AC2 BIT 0 = 1
     0002   7101   A              CAI    AC1,1         ;2'S COMPLEMENT AC1
     0003   6840   A              RADD   AC1,AC0       ;AC1 + AC0 -> AC0
     0004   4103   A              BOC    1,EXIT        ;EXIT IF AC0 = AC1
     0005   5202   A              LI     AC2,2         ;SET AC2 BIT 1 = 1
     0006   4A01   A              BOC    10,EXIT       ;EXIT IF AC0 > AC1
     0007   5204   A              LI     AC2,4         ;SET AC2 BIT 2 = 1
     0008   5C80   A     EXIT:    RCPY   AC2,AC0       ;COPY AC2 TO AC0
     0009   6600   A              PULL   AC2           ;RESTORE AC2 FROM STACK
     000A   8000   A              RTS                  ;RETURN
            0000                  .END
```

MSI DM54/DM74160A,LS160,161A,LS161,162A,LS162,163A,LS163

Synchronous 4-Bit Counters

General Description

These synchronous, presettable counters feature an internal carry look-ahead for application in high-speed counting designs. The 160A, 162A, LS160, LS162, are decade counters and the 161A, 163A, LS161, LS163 are 4-bit binary counters. The carry output is decoded by means of a NOR gate, thus preventing spikes during the normal counting mode of operation. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincident with each other when so instructed by the count-enable inputs and internal gating. This mode of operation eliminates the output counting spikes which are normally associated with asynchronous (ripple clock) counters. A buffered clock input triggers the four flip-flops on the rising (positive-going) edge of the clock input waveform.

These counters are fully programmable; that is, the outputs may be preset to either level. As presetting is synchronous, setting up a low level at the load input disables the counter and causes the outputs to agree with the setup data after the next clock pulse regardless of the levels of the enable input. Low-to-high transitions at the load input of the 160A through 163A or LS160 through LS163 are perfectly acceptable, regardless of the logic levels on the clock or enable inputs. The clear function for the 160A, 161A, LS160, and LS161 is asynchronous; and a low level at the clear input sets all four of the flip-flop outputs low regardless of the levels of clock, load, or enable inputs. The clear function for the 162A, 163A, LS162, LS163, is synchronous; and a low level at the clear input sets all four of the flip-flop outputs low after the next clock pulse, regardless of the levels of the enable inputs. This synchronous clear allows the count length to be modified easily, as decoding the maximum count desired can be accomplished with one external NAND gate. The gate output is connected to the clear input to synchronously clear the counter to all low outputs. Low-to-high transitions at the clear input of the 162A and 163A are also permissible regardless of the logic levels on the clock, enable, or load inputs.

The carry look-ahead circuitry provides for cascading counters for n-bit synchronous applications without additional gating. Instrumental in accomplishing this function are two count-enable inputs and a ripple carry output. Both count-enable inputs (P and T) must be high to count, and input T is fed forward to enable the ripple carry output. The ripple carry output thus enabled will produce a high-level output pulse with a duration approximately equal to the high-level portion of the Q_A output. This high-level overflow ripple carry pulse can be used to enable successive cascaded stages. High-to-low-level transitions at the enable P or T inputs of the 160A through 163A or LS160 through LS163, may occur regardless of the logic level on the clock.

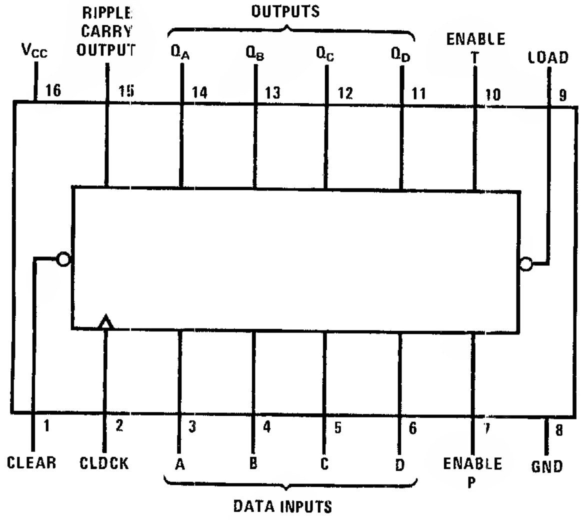
LS160 through LS163 feature a fully independent clock circuit. Changes made to control inputs (enable P or T, load or clear) that will modify the operating mode have no effect until clocking occurs. The function of the counter (whether enabled, disabled, loading, or counting) will be dictated solely by the conditions meeting the stable setup and hold times.

Features

- Synchronously programmable
- Internal look-ahead for fast counting
- Carry output for n-bit cascading
- Synchronous counting
- Load control line
- Diode-clamped inputs

TYPE	TYPICAL PROPAGATION TIME, CLOCK TO Q OUTPUT	TYPICAL CLOCK FREQUENCY	TYPICAL POWER DISSIPATION
160 thru 163	14 ns	35 MHz	315 mW
LS160 thru LS163	14 ns	32 MHz	93 mW

Connection Diagram



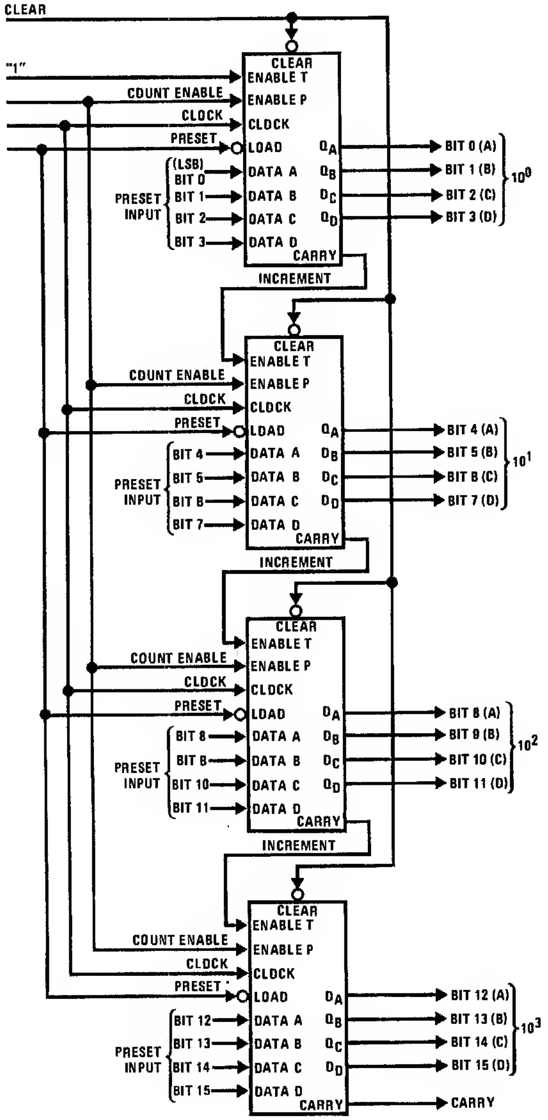
54160A(J), (W); 74160A(J), (N), (W);
54LS160/74LS160(J), (N), (W);
54161A(J), (W); 74161A(J), (N), (W);
54LS161/74LS161(J), (N), (W);
54162A(J), (W); 74162A(J), (N), (W);
54LS162/74LS162(J), (N), (W);
54163A(J), (W); 74163A(J), (N), (W);
54LS163/74LS163(J), (N), (W)

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows how four DM74160/DM74162 devices may be cascaded to form a fully synchronous 4-stage BCD counter. For this application, counting is enabled when a high Count Enable signal is applied to the E/P inputs of the counter stages. While counting is enabled, the look-ahead carry (C/O) output of each stage serves as a gated count enable signal to the next stage to allow each stage to be incremented at the same time that the previous stage is clocked to zero. Thus, a high look-ahead carry output is provided by the last stage when the counter is at the maximum value of 9999.

DM74160/DM74162 BCD Counter



ASSIGNMENTS

The 4-stage BCD counter function may be implemented with PACE as a multiple-entry subroutine. The flowchart and program listing that follow assume that a memory location is dedicated to storage of the count, that AC0 is used as a working register for altering the stored count, and that input/output assignments are as listed below.

INPUTS:

DM74160/ DM74162	PACE
Clear	CLEAR entry to Decade Counter subroutine
Load	PRESET entry to Decade Counter subroutine
Count (E/P, E/T, CK)	INCREMENT entry to Decade Counter subroutine (clock rate is equal to frequency of calling)

OUTPUTS:

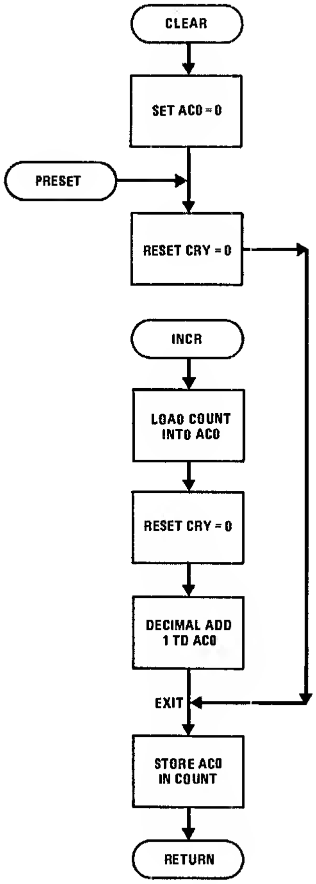
DM74160/ DM74162	PACE
0000-9999	Contents of memory location COUNT
C/O (last stage)	Status Register bit 7 (carry flag)

FUNCTIONAL OPERATION

This program is written as a multiple-entry subroutine that clears, presets, or increments a BCD counter. When the subroutine is entered at the CLEAR address, the contents of AC0 are set to zero, the carry flag is reset to clear any previous status (see the preface) and the contents of AC0 are loaded into memory location COUNT to initialize the stored value to zero. When the subroutine is entered at the PRESET address, it is assumed that the desired preset value has already been loaded into AC0 by the main program so the contents of AC0 are not altered during execution of the subroutine. Thus, after the carry flag is reset the contents of AC0 are loaded into COUNT to initialize the stored count to some value between 0000_{10} and 9999_{10}.

The INCREMENT entry to the subroutine combines the functions of the E/P, E/T, and CK inputs and the C/O output of the DM74160/DM74162 counters. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0 and the carry flag is reset. The contents of AC0 are then incremented by one via a Decimal Add (DECA) instruction, and the new value is returned to COUNT. Use of the Decimal Add instruction allows the stored count to be treated as a 4-digit decimal number and the carry flag to be set when AC0 is incremented from 9999_{10} to 0000_{10}. Since the subroutine is otherwise exited with the carry flag reset, the carry flag can be tested upon return to the main program to detect completion of a normal count sequence.

FLOW CHART



Clear AC0

Clear carry

Load the previous count

Clear carry

Add 1 to the count

Save the count

PROGRAM LISTING

```
                 ;         BCD COUNTER
      ØØØØ       ACØ       =       Ø
      ØØØ7       CRY       =       7               ;CARRY
ØØØØ  5ØØØ  A    CLEAR:    LI      ACØ,Ø           ;SET ACØ = Ø
ØØØ1  37ØØ  A    PRESET:   PFLG    CRY             ;SET CARRY = Ø
ØØØ2  19Ø3  A              JMP     EXIT            ;
ØØØ3  C1Ø4  A    INCR:     LD      ACØ,COUNT       ;LOAD COUNT INTO ACØ
ØØØ4  37ØØ  A              PFLG    CRY             ;SET CARRY = Ø
ØØØ5  89Ø3  A              DECA    ACØ,ONE         ;DECIMAL ADD 1 TO ACØ
ØØØ6  D1Ø1  A    EXIT:     ST      ACØ,COUNT       ;STORE ACØ IN COUNT
ØØØ7  8ØØØ  A              RTS                     ;RETURN
ØØØ8  ØØØØ  A    COUNT:    .WORD   Ø               ;COUNTER SAVE
ØØØ9  ØØØ1  A    ONE:      .WORD   1               ;CONSTANT
      ØØØØ                 .END
```

MSI DM54/DM74160A,LS160,161A,LS161,162A,LS162,163A,LS163

Synchronous 4-Bit Counters

General Description

These synchronous, presettable counters feature an internal carry look-ahead for application in high-speed counting designs. The 160A, 162A, LS160, LS162, are decade counters and the 161A, 163A, LS161, LS163 are 4-bit binary counters. The carry output is decoded by means of a NOR gate, thus preventing spikes during the normal counting mode of operation. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincident with each other when so instructed by the count-enable inputs and internal gating. This mode of operation eliminates the output counting spikes which are normally associated with asynchronous (ripple clock) counters. A buffered clock input triggers the four flip-flops on the rising (positive-going) edge of the clock input waveform.

These counters are fully programmable; that is, the outputs may be preset to either level. As presetting is synchronous, setting up a low level at the load input disables the counter and causes the outputs to agree with the setup data after the next clock pulse regardless of the levels of the enable input. Low-to-high transitions at the load input of the 160A through 163A or LS160 through LS163 are perfectly acceptable, regardless of the logic levels on the clock or enable inputs. The clear function for the 160A, 161A, LS160, and LS161 is asynchronous; and a low level at the clear input sets all four of the flip-flop outputs low regardless of the levels of clock, load, or enable inputs. The clear function for the 162A, 163A, LS162, LS163, is synchronous; and a low level at the clear input sets all four of the flip-flop outputs low after the next clock pulse, regardless of the levels of the enable inputs. This synchronous clear allows the count length to be modified easily, as decoding the maximum count desired can be accomplished with one external NAND gate. The gate output is connected to the clear input to synchronously clear the counter to all low outputs. Low-to-high transitions at the clear input of the 162A and 163A are also permissible regardless of the logic levels on the clock, enable, or load inputs.

The carry look-ahead circuitry provides for cascading counters for n-bit synchronous applications without additional gating. Instrumental in accomplishing this function are two count-enable inputs and a ripple carry output. Both count-enable inputs (P and T) must be high to count, and input T is fed forward to enable the ripple carry output. The ripple carry output thus enabled will produce a high-level output pulse with a duration approximately equal to the high-level portion of the Q_A output. This high-level overflow ripple carry pulse can be used to enable successive cascaded stages. High-to-low-level transitions at the enable P or T inputs of the 160A through 163A or LS160 through LS163, may occur regardless of the logic level on the clock.

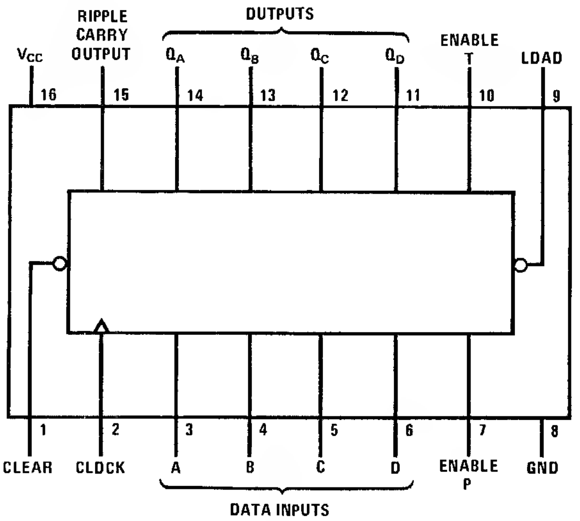
LS160 through LS163 feature a fully independent clock circuit. Changes made to control inputs (enable P or T, load or clear) that will modify the operating mode have no effect until clocking occurs. The function of the counter (whether enabled, disabled, loading, or counting) will be dictated solely by the conditions meeting the stable setup and hold times.

Features

- Synchronously programmable
- Internal look-ahead for fast counting
- Carry output for n-bit cascading
- Synchronous counting
- Load control line
- Diode-clamped inputs

TYPE	TYPICAL PROPAGATION TIME, CLOCK TO Q OUTPUT	TYPICAL CLOCK FREQUENCY	TYPICAL POWER DISSIPATION
160 thru 163	14 ns	35 MHz	315 mW
LS160 thru LS163	14 ns	32 MHz	93 mW

Connection Diagram



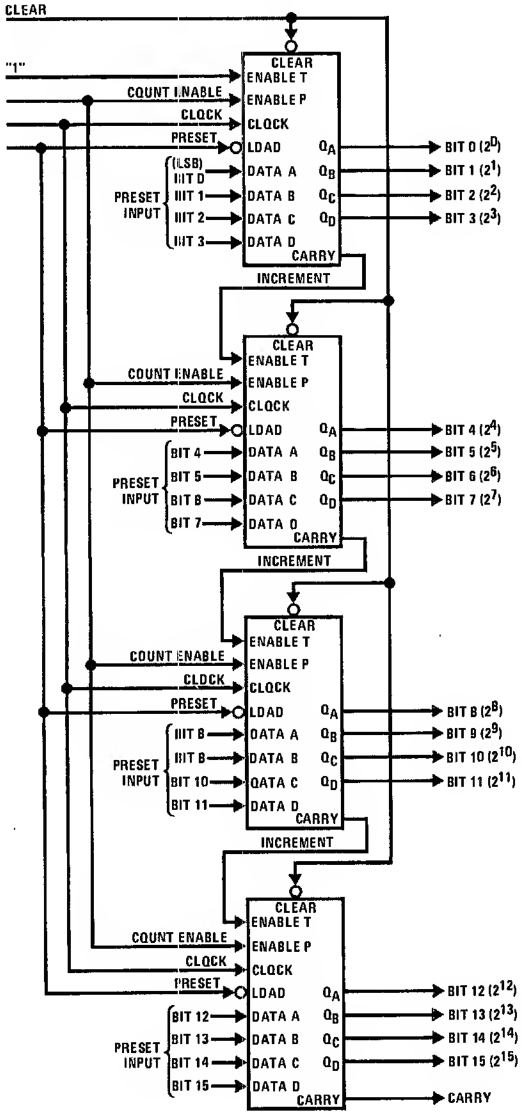
54160A(J), (W); 74160A(J), (N), (W);
54LS160/74LS160(J), (N), (W);
54161A(J), (W); 74161A(J), (N), (W);
54LS161/74LS161(J), (N), (W);
54162A(J), (W); 74162A(J), (N), (W);
54LS162/74LS162(J), (N), (W);
54163A(J), (W); 74163A(J), (N), (W);
54LS163/74LS163(J), (N), (W)

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows how four DM74161/DM74163 devices may be cascaded to form a fully synchronous 16-bit binary counter. For this application, counting is enabled when a high Count Enable signal is applied to the E/P inputs of the counter stages. While counting is enabled, the look-ahead carry (C/O) output of each stage serves as a gated count enable signal to the next stage to allow each stage to be incremented at the same time that the previous stage is clocked to zero. Thus, a high look-ahead carry output is provided by the last stage when the counter is at the maximum value of FFFF.

DM74161/DM74163 Binary Counter



ASSIGNMENTS

The 16-bit binary counter function may be implemented with PACE as a multiple-entry subroutine. The flowchart and program listing that follow assume that a memory location is dedicated to storage of the count, that AC0 is used as a working register for altering the stored count, and that input/output assignments are as listed below.

INPUTS:

DM74161/ DM74163	PACE
Clear	CLEAR entry to Binary Counter subroutine
Load	PRESET entry to Binary Counter subroutine
Count (E/P) E/T, CK)	INCREMENT entry to Binary Counter subroutine (clock rate is equal to frequency of calling)

OUTPUTS:

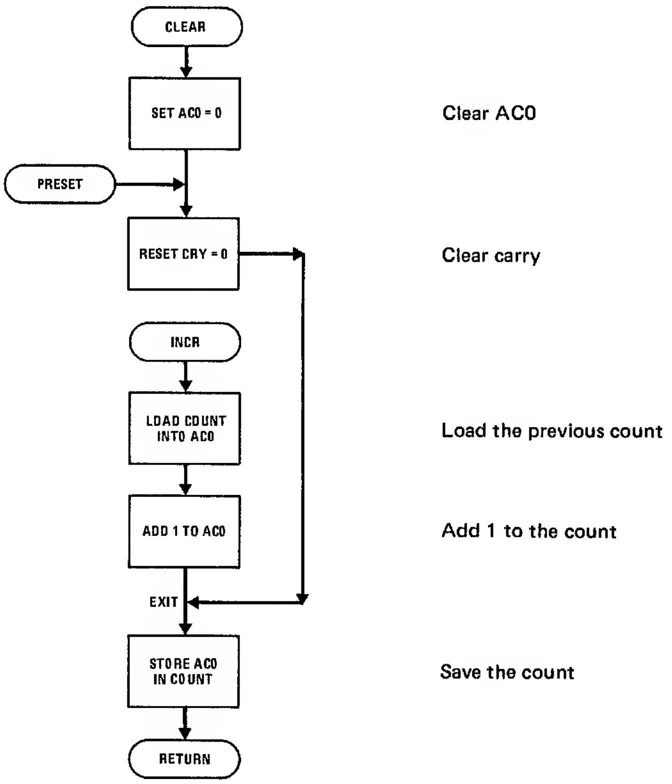
DM74161/ DM74163	PACE
0000-9999	Contents of memory location COUNT
C/O (last stage)	Status Register bit 7 (carry flag)

FUNCTIONAL OPERATION

This program is written as a multiple-entry subroutine that clears, presets, or increments a binary counter. When the subroutine is entered at the CLEAR address, the contents of AC0 are set to zero, the carry flag is reset to clear any previous status (see the preface), and the contents of AC0 are loaded into memory location COUNT to initialize the stored value to zero. When the subroutine is entered at the PRESET address, it is assumed that the desired preset value has already been loaded into AC0 by the main program so the contents of AC0 are not altered during execution of the subroutine. Thus, after the carry flag is reset the contents of AC0 are loaded into COUNT to initialize the stored count to some value between 0000 and FFFF.

The INCREMENT entry to the subroutine combines the functions of the E/P, E/T, and CK inputs and the C/O output of the DM74161/DM74163 counters. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0, then the contents of AC0 are incremented by one via an ADD instruction, and the new value is returned to COUNT. Use of the ADD instruction allows the stored count to be treated as a 16-bit binary number and the carry flag to be set when AC0 is incremented from FFFF to 0000. Since the carry flag is automatically reset by the other two entries to the subroutine, it can be tested upon return to the main program to detect completion of a normal count sequence.

FLOW CHART



PROGRAM LISTING

```
                     ;        BINARY  COUNTER
        ØØØØ         ACØ      =       Ø
  ØØØØ  5ØØØ   A     CLEAR:   LI      ACØ,Ø            ;SET ACØ = Ø
  ØØØ1  37ØØ   A     PRESET:  PFLG    7                ;SET CARRY = Ø
  ØØØ2  19Ø2   A              JMP     EXIT             ;
  ØØØ3  C1Ø3   A     INCR:    LD      ACØ,COUNT        ;LOAD COUNT INTO ACØ
  ØØØ4  E1Ø3   A              ADD     ACØ,ONE          ;ADD 1 TO COUNT IN ACØ
  ØØØ5  D1Ø1   A     EXIT:    ST      ACØ,COUNT        ;STORE ACØ IN COUNT
  ØØØ6  8ØØØ   A              RTS                      ;RETURN
  ØØØ7  ØØØØ   A     COUNT:   .WORD   Ø                ;COUNTER SAVE
  ØØØ8  ØØØ1   A     ONE:     .WORD   1                ;CONSTANT
        ØØØØ                  .END
```

MSI DM54/DM74160A,LS160,161A,LS161,162A,LS162,163A,LS163

Synchronous 4-Bit Counters

General Description

These synchronous, presettable counters feature an internal carry look-ahead for application in high-speed counting designs. The 160A, 162A, LS160, LS162, are decade counters and the 161A, 163A, LS161, LS163 are 4-bit binary counters. The carry output is decoded by means of a NOR gate, thus preventing spikes during the normal counting mode of operation. Synchronous operation is provided by having all flip-flops clocked simultaneously so that the outputs change coincident with each other when so instructed by the count-enable inputs and internal gating. This mode of operation eliminates the output counting spikes which are normally associated with asynchronous (ripple clock) counters. A buffered clock input triggers the four flip-flops on the rising (positive-going) edge of the clock input waveform.

These counters are fully programmable; that is, the outputs may be preset to either level. As presetting is synchronous, setting up a low level at the load input disables the counter and causes the outputs to agree with the setup data after the next clock pulse regardless of the levels of the enable input. Low-to-high transitions at the load input of the 160A through 163A or LS160 through LS163 are perfectly acceptable, regardless of the logic levels on the clock or enable inputs. The clear function for the 160A, 161A, LS160, and LS161 is asynchronous; and a low level at the clear input sets all four of the flip-flop outputs low regardless of the levels of clock, load, or enable inputs. The clear function for the 162A, 163A, LS162, LS163, is synchronous; and a low level at the clear input sets all four of the flip-flop outputs low after the next clock pulse, regardless of the levels of the enable inputs. This synchronous clear allows the count length to be modified easily, as decoding the maximum count desired can be accomplished with one external NAND gate. The gate output is connected to the clear input to synchronously clear the counter to all low outputs. Low-to-high transitions at the clear input of the 162A and 163A are also permissible regardless of the logic levels on the clock, enable, or load inputs.

The carry look-ahead circuitry provides for cascading counters for n-bit synchronous applications without additional gating. Instrumental in accomplishing this function are two count-enable inputs and a ripple carry output. Both count-enable inputs (P and T) must be high to count, and input T is fed forward to enable the ripple carry output. The ripple carry output thus enabled will produce a high-level output pulse with a duration approximately equal to the high-level portion of the Q_A output. This high-level overflow ripple carry pulse can be used to enable successive cascaded stages. High-to-low-level transitions at the enable P or T inputs of the 160A through 163A or LS160 through LS163, may occur regardless of the logic level on the clock.

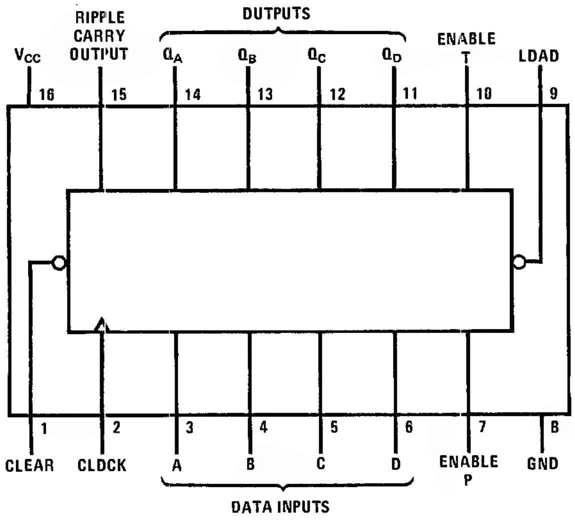
LS160 through LS163 feature a fully independent clock circuit. Changes made to control inputs (enable P or T, load or clear) that will modify the operating mode have no effect until clocking occurs. The function of the counter (whether enabled, disabled, loading, or counting) will be dictated solely by the conditions meeting the stable setup and hold times.

Features

- Synchronously programmable
- Internal look-ahead for fast counting
- Carry output for n-bit cascading
- Synchronous counting
- Load control line
- Diode-clamped inputs

TYPE	TYPICAL PROPAGATION TIME, CLOCK TO Q OUTPUT	TYPICAL CLOCK FREQUENCY	TYPICAL POWER DISSIPATION
160 thru 163	14 ns	35 MHz	315 mW
LS160 thru LS163	14 ns	32 MHz	93 mW

Connection Diagram



54160A(J), (W); 74160A(J), (N), (W);
54LS160/74LS160(J), (N), (W);
54161A(J), (W); 74161A(J), (N), (W);
54LS161/74LS161(J), (N), (W);
54162A(J), (W); 74162A(J), (N), (W);
54LS162/74LS162(J), (N), (W);
54163A(J), (W); 74163A(J), (N), (W);
54LS163/74LS163(J), (N), (W)

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The circuit diagram shows a 16-bit binary counter interconnected with a 5-stage decade counter to form a Binary-to-BCD converter. (Basic operation of the binary and BCD counters is covered on pages 4-34 to 4-36 and 4-31 to 4-33.) For this application, operation of the counters is controlled by the three flip-flops that process the Clock, Start Conversion, and Carry signals to enable each conversion cycle. As shown in the timing diagram, each conversion cycle is initiated when the Load flip-flop is preset on the leading-edge of the Start Conversion pulse, which enters the complemented binary input into the binary counter and enters the starting value 0000 into the decade counter. The Load flip-flop then remains set until it is clocked reset on the first negative alternation of the clock following termination of the Start Conversion pulse. When the Load flip-flop is reset, the low Q output sets the Delay flip-flop, and the resulting high Conversion Control signal is clocked into the Start/Stop flip-flop on the positive alternation of the clock, which allows counting to start one clock pulse later.

While the Q output of the Start/Stop flip-flop is high, both the binary and decade counters are counted up by the clock input until the binary counter provides a look-ahead Carry output at the count of FFFF. The look-ahead Carry then resets the Delay flip-flop, and the resulting low Conversion Control signal is clocked into the Start/Stop flip-flop on the next positive alternation of the clock to terminate the conversion cycle. Thus, the conversion cycle is terminated with the output of the binary counter equal to 0000 and the output of the decade counter equal to the decimal value of the original binary input.

ASSIGNMENTS

The flowchart and program listing that follow assume that an 8-address block of memory is dedicated to storage of a binary-to-BCD conversion table, that AC0 is used as an input data register for entry of the 16-bit binary input, that all four accumulators are used as working registers during performance of the conversion, and that the resulting BCD output is provided via AC0 (four least-significant digits) and AC1 (most-significant digit).

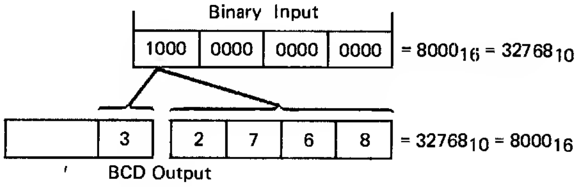
FUNCTIONAL DESCRIPTION

This routine uses a look-up table to perform the binary-to-BCD conversion. Each bit in a 16-bit binary number has a decimal value, as shown in the table below.

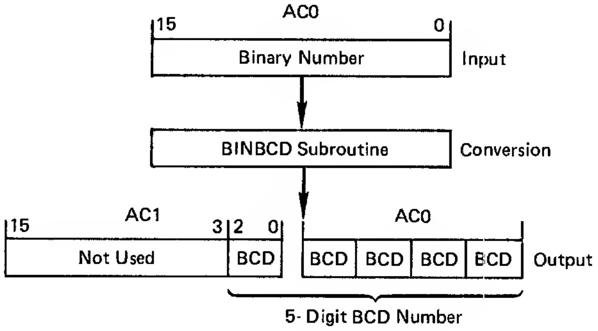
BINARY BIT	BCD VALUE (if bit set)	BINARY BIT	BCD VALUE (if bit set)
0	1	8	256
1	2	9	512
2	4	10	1024
3	8	11	2048
4	16	12	4096
5	32	13	8192
6	64	14	16384
7	128	15	32768

If a bit is set in the binary number, its BCD value is added decimally to the contents of a register (the less-significant register). Bits 0 through 12 of the binary number are straight look-ups, but bits 13 through 15 require additional operations. Bit 13 may generate carry; if so, a 1 is added to the contents of a second register (the most-significant register). The BCD values for bits 14 and 15 are too large for the less-significant register, so the most-significant BCD digit for bits 14 and 15 is added to the contents of the most-significant register.

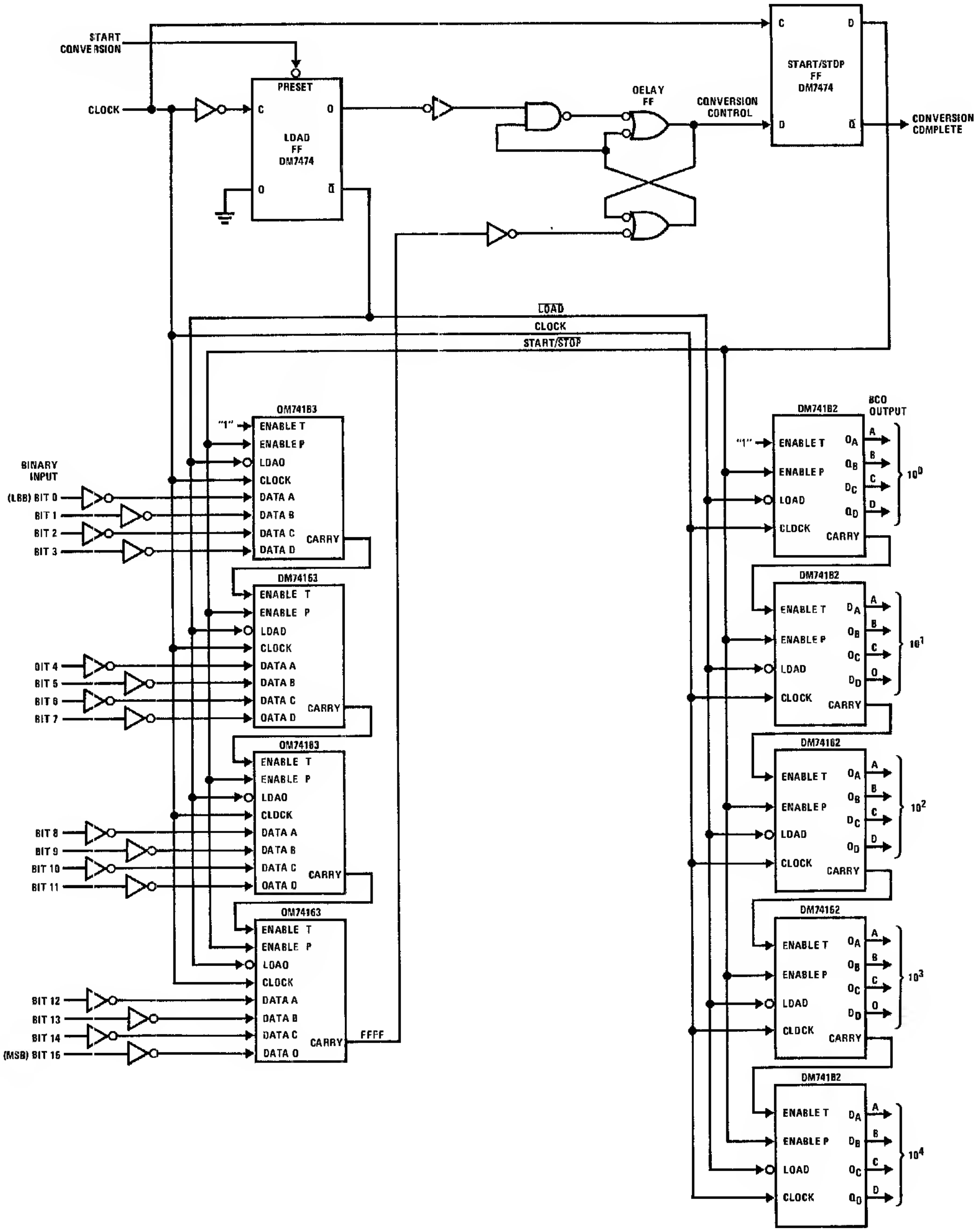
The example below shows the conversion of bit 15.



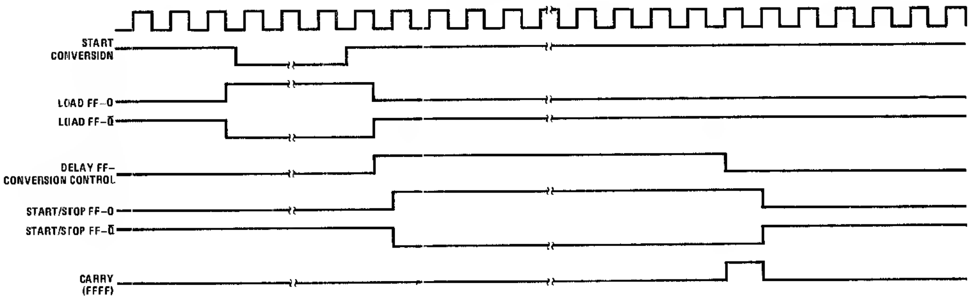
The Binary-to-BCD (BINBCD) subroutine is entered with the binary number to be converted in AC0. The results of the conversion are returned in AC1 and AC0. AC1 contains the most-significant BCD number and AC0 contains the four less-significant BCD numbers. The figure below illustrates the operation of the routine.



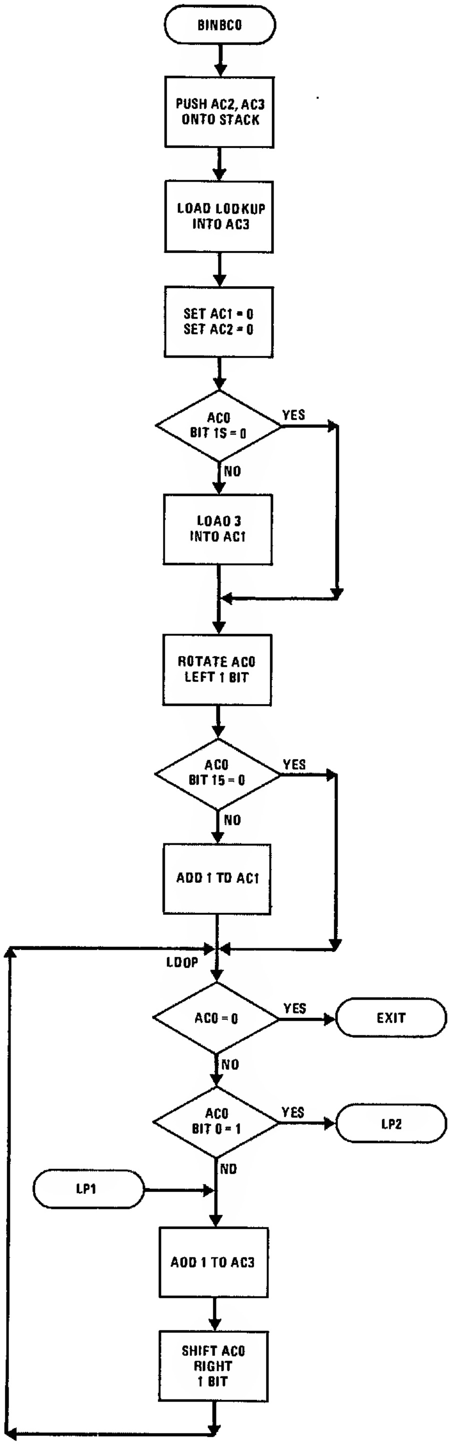
Upon entering the BINBCD routine, AC2 and AC3 are saved on the stack, the address of the look-up table is loaded into AC3, and AC1 and AC2 are cleared. AC1 and AC2 will contain the BCD sum during conversion. (AC1 contains the most-significant BCD digit.) Next, binary input bit 15 in AC0 is tested. If it equals one, a three is loaded into AC1. AC0 is rotated left one position and input bit 14 is tested. If it equals one, a one is added to AC1. The program then goes into a loop, first checking if AC0 equals zero. If AC0 does not equal zero, bit 0 of AC0 is tested. If it equals one, AC0 and AC2 are exchanged, the BCD value for the bit is added decimally to AC0, carry is tested, and if high a one is added to AC1. Finally, AC0 and AC2 are again exchanged. The loop is completed by incrementing the look-up table pointer by one, shifting AC0 right one position, then branching to the beginning of the loop to test the next bit. If AC0 equals zero, the conversion is completed, and the program jumps to exit. Exit copies the less-significant four BCD digits from AC2 to AC0, restores AC2 and AC3 from the stack, and returns.



Timing Diagram



FLOW CHART



Save contents of AC2 and AC3

Load address of look-up table

Clear AC1 and AC2

Bit 15 of input = 0?

Yes, Load 3 into AC1, 2768 will be added to AC0 to make 32768 (the BCD value of Bit 15)

Position input bit 14

Input bit 14 = 0?

Yes, add 1 to AC1, 6384 will be added to AC0 to make 16384 (the BCD value of bit 14)

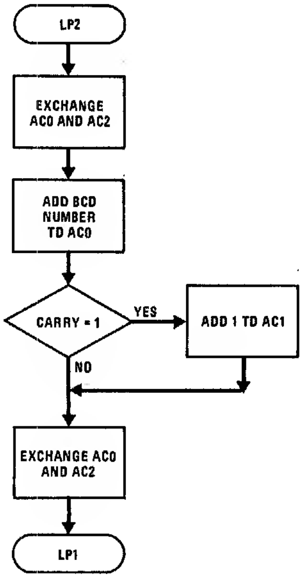
Exit if AC0 = 0

Is the input bit 1?

Increment look-up table pointer

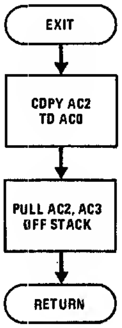
Shift AC0 right 1 position to test next bit

FLOW CHART (Continued)



Add number from look-up table to BCD count

Add 1 to AC1 if the carry is set



BCD number is now in AC1 and AC0

Restore AC2 and AC3

PROGRAM LISTING

```
                   ;         BINARY  TO  BCD
          0000     AC0       =       0
          0001     AC1       =       1
          0002     AC2       =       2
          0003     AC3       =       3
          000A     CARRY     =       10
0000 6200 A        BINBCD:   PUSH    AC2             ;SAVE REGISTERS ON STACK
0001 6300 A                  PUSH    AC3             ;
0002 CD17 A                  LD      AC3,LOOKUP      ;LOAD ADDRESS OF LOOKUP
0003 5100 A                  LI      AC1,0           ;CLEAR AC1
0004 5200 A                  LI      AC2,0           ;CLEAR AC2
0005 4201 A                  BOC     2,.+2           ;BRANCH IF AC0 BIT 15=0
0006 5103 A                  LI      AC1,3           ;LOAD 3 INTO AC1
0007 2002 A                  ROL     AC0,1,0         ;ROTATE AC0 LEFT 1 BIT
0008 4201 A                  BOC     2,LOOP          ;BRANCH IF AC0 BIT 15=0
0009 7901 A                  AISZ    AC1,1           ;ADD 1 TO AC1
000A 410B A        LOOP:     BOC     1,EXIT          ;BRANCH IF AC0 = 0
000B 4303 A                  BOC     3,LP2           ;BRANCH IF AC0 BIT 0 = 1
000C 7B01 A        LP1:      AISZ    AC3,1           ;INCREMENT TABLE POINTER
000D 2C02 A                  SHR     AC0,1,0         ;SHIFT AC0 RIFHT 1 BIT
000E 19FB A                  JMP     LOOP            ;CONTINUE TESTING
000F 6E00 A        LP2:      RXCH    AC0,AC2         ;EXCHANGE AC0 AND AC2
0010 8B00 A                  DECA    AC0,0(AC3)      ;ADD BCD NUMBER TO AC0
0011 4A02 A                  BOC     CARRY,CRYHI     ;BRANCH IF CARRY = 1
0012 6E00 A        LP3:      RXCH    AC0,AC2         ;EXCHANGE AC0 AND AC2
0013 19F8 A                  JMP     LP1             ;
0014 7901 A        CRYHI:    AISZ    AC1,1           ;ADD 1 TO AC1
0015 19FC A                  JMP     LP3             ;
0016 5C80 A        EXIT:     RCPY    AC2,AC0         ;COPY AC2 TO AC0
0017 6700 A                  PULL    AC3             ;RESTORE REGISTERS
0018 6600 A                  PULL    AC2             ;
0019 8000 A                  RTS                     ;RETURN
001A 001B T        LOOKUP:   .WORD   .+1             ;LOOKUP TABLE
001B 2768 A                  .WORD   02768           ;BIT 15
001C 0001 A                  .WORD   00001           ;BIT 0
001D 0002 A                  .WORD   00002           ;BIT 1
001E 0004 A                  .WORD   00004           ;BIT 2
001F 0008 A                  .WORD   00008           ;BIT 3
0020 0016 A                  .WORD   00016           ;BIT 4
0021 0032 A                  .WORD   00032           ;BIT 5
0022 0064 A                  .WORD   00064           ;BIT 6
0023 0128 A                  .WORD   00128           ;BIT 7
0024 0256 A                  .WORD   00256           ;BIT 8
0025 0512 A                  .WORD   00512           ;BIT 9
0026 1024 A                  .WORD   01024           ;BIT 10
0027 2048 A                  .WORD   02048           ;BIT 11
0028 4096 A                  .WORD   04096           ;BIT 12
0029 8192 A                  .WORD   08192           ;BIT 13
002A 6384 A                  .WORD   06384           ;BIT 14
          0000               .END
```

DM54/DM7483,LS83A,LS283

4-Bit Binary Adders with Fast Carry

General Description

These full adders perform the addition of two 4-bit binary numbers. The sum (Σ) outputs are provided for each bit and the resultant carry (C4) is obtained from the fourth bit. These adders feature full internal look ahead across all four bits. This provides the system designer with partial look-ahead performance at the economy and reduced package count of a ripple-carry implementation.

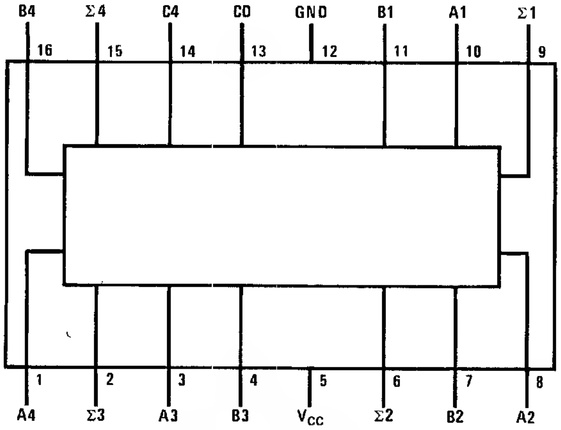
The adder logic, including the carry, is implemented in its true form meaning that the end-around carry can be accomplished without the need for logic or level inversion.

Features

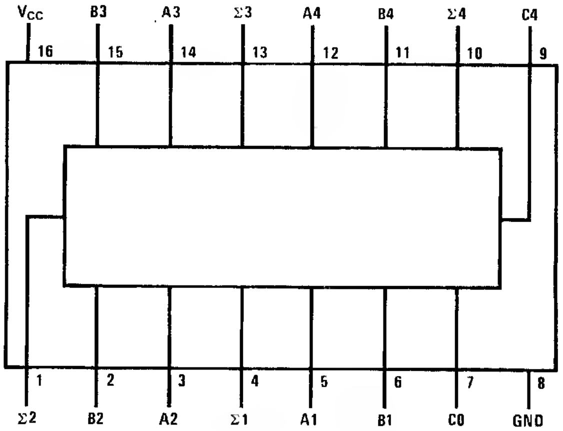
- Full-carry look-ahead across the four bits
- Systems achieve partial look-ahead performance with the economy of ripple carry

TYPE	TYPICAL ADD TIMES TWO 8-BIT WORDS	TYPICAL ADD TIMES TWO 16-BIT WORDS	TYPICAL POWER DISSIPATION PER 4-BIT ADDER
83	23 ns	43 ns	290 mW
LS83A	25 ns	45 ns	95 mW
LS283	25 ns	45 ns	95 mW

Connection Diagrams and Truth Table



5483(J), (W); 7483(J), (N), (W);
54LS83A/74LS83A(J), (N), (W)



54LS283/74LS283(J), (N), (W)

INPUT				OUTPUT					
				WHEN C0 = L / WHEN C2 = L			WHEN C0 = H / WHEN C2 = H		
A1 / A3	B1 / B3	A2 / A4	B2 / B4	Σ1 / Σ3	Σ2 / Σ4	C2 / C4	Σ1 / Σ3	Σ2 / Σ4	C2 / C4
L	L	L	L	L	L	L	H	L	L
H	L	L	L	H	L	L	L	H	L
L	H	L	L	H	L	L	L	H	L
H	H	L	L	L	H	L	H	H	L
L	L	H	L	L	H	L	H	H	L
H	L	H	L	H	H	L	L	L	H
L	H	H	L	H	H	L	L	L	H
H	H	H	L	L	L	H	H	L	H
L	L	L	H	L	H	L	H	H	L
H	L	L	H	H	H	L	L	L	H
L	H	L	H	H	H	L	L	L	H
H	H	L	H	L	L	H	H	L	H
L	L	H	H	L	L	H	H	L	H
H	L	H	H	H	L	H	L	H	H
L	H	H	H	H	L	H	L	H	H
H	H	H	H	L	H	H	H	H	H

H = High Level, L = Low Level

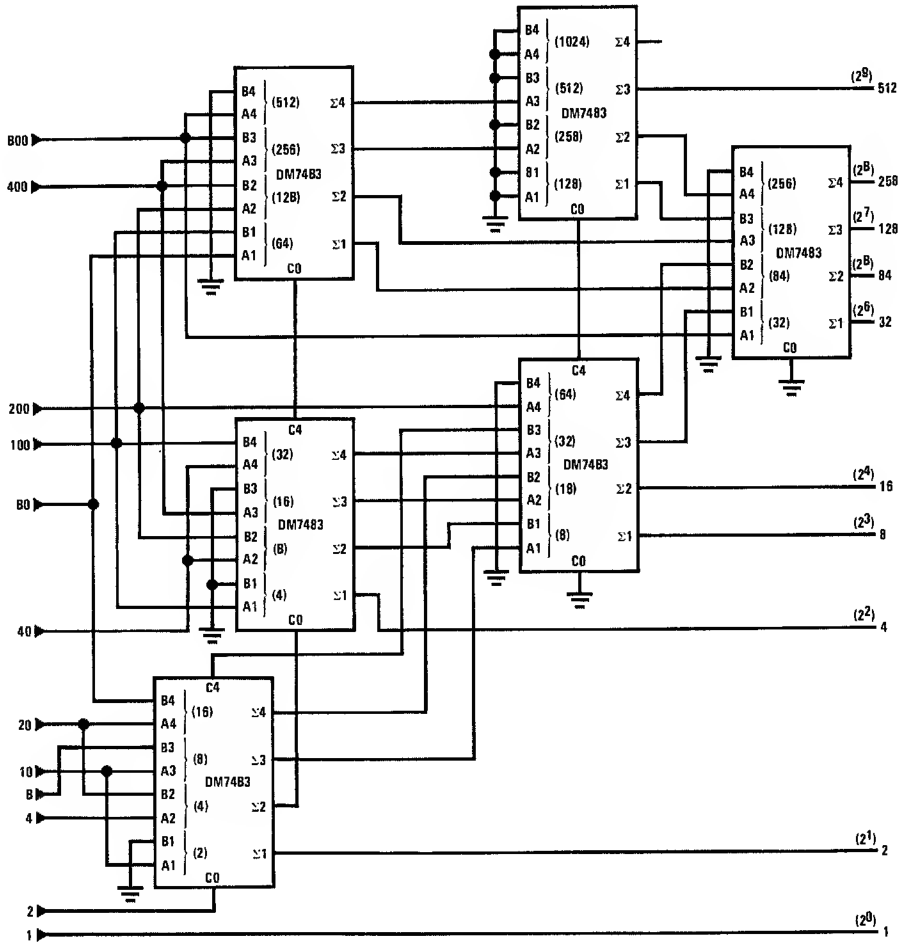
Note : Input conditions at A1, B1, A2, B2, and C0 are used to determine outputs Σ1 and Σ2 and the value of the internal carry C2. The values at C2, A3, B3, A4, and B4 are then used to determine outputs Σ3, Σ4, and C4.

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

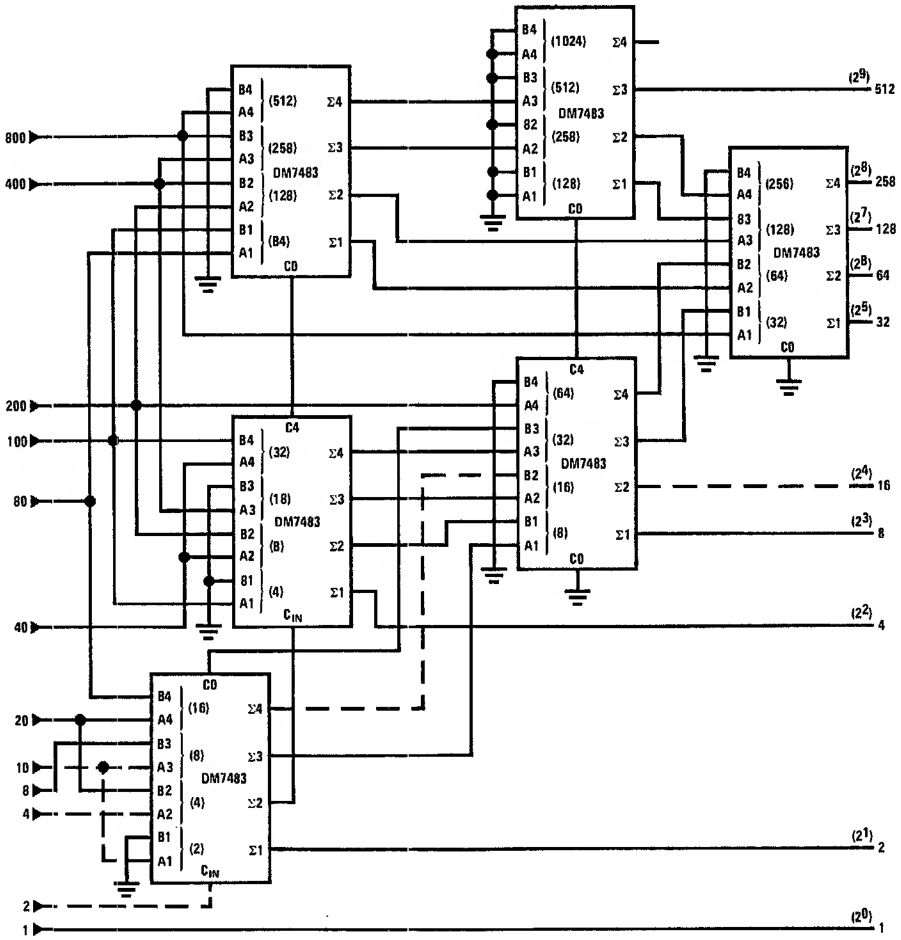
The diagram below illustrates five, 4-bit full adders connected to convert a 3-digit BCD input to a 10-bit binary output. A 3-digit BCD value was chosen for this application because it is sufficiently large to illustrate overall circuit principles of operation, yet does not require an excessively complex logic diagram. (Cascading of the adder stages to encompass a 5-digit BCD input is readily accomplished, but would increase the number of adders required by a factor of ten.) In the two examples provided to illustrate circuit operation, the dashed lines indicate a logic one state and the solid lines indicate a logic zero state. The method used for the conversion separates each power of ten into its binary equivalent, then sums these individual binary values to derive the final result.

BCD-to-Binary



Example 1. BCD-16 to Binary

$$16 = \left\{\begin{array}{l} 10\left\{\begin{array}{l} 8 \\ +2 \end{array}\right. \\ +6\left\{\begin{array}{l} +4 \\ +2 \end{array}\right. \end{array}\right. \begin{array}{cr} = & 1\,0\,0\,0 \\ = & 0\,0\,1\,0 \\ = & 0\,1\,0\,0 \\ = & 0\,0\,1\,0 \\ \hline & 1\,0\,0\,0\,0\ (2^4) \end{array}$$

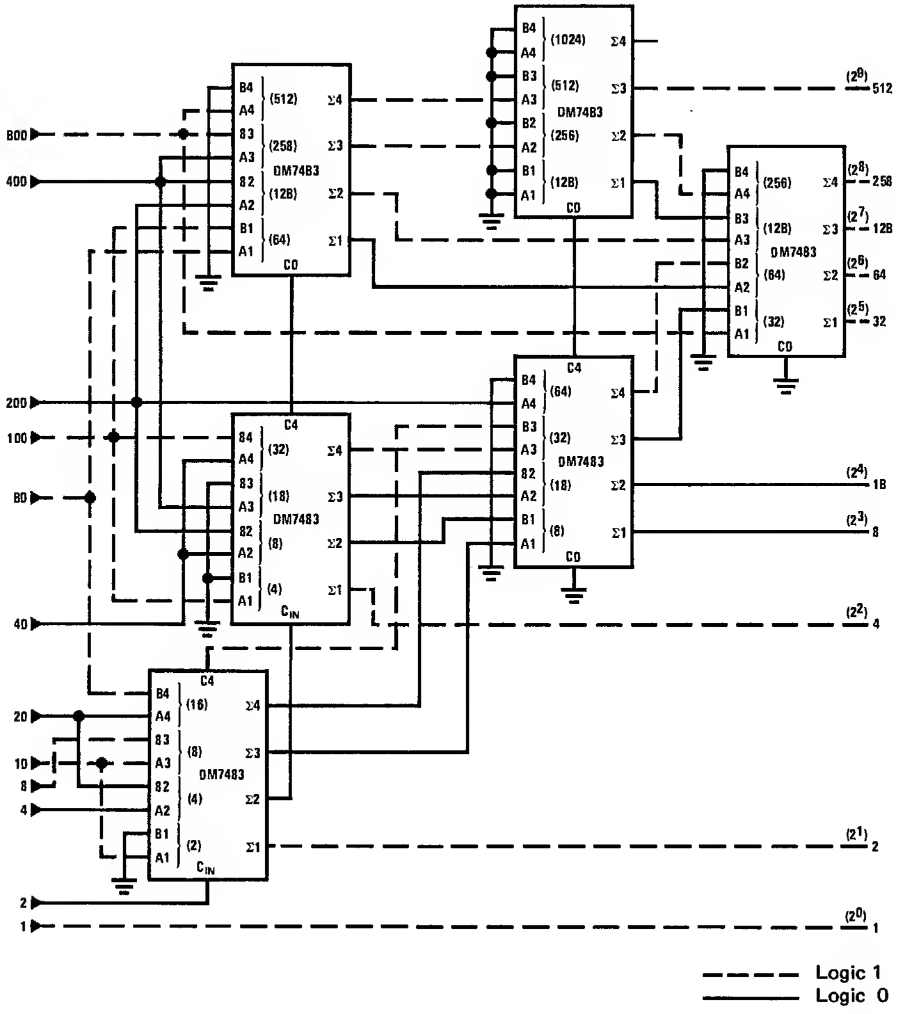


– – – – Logic 1

——— Logic 0

Example 2. BCD-999 to Binary

$$\begin{array}{l}
999 = \begin{cases} 900 = \begin{cases} 800 \begin{cases} 512 = 1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 245 = 0\,1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 32 = 0\,0\,0\,0\,1\,0\,0\,0\,0\,0 \end{cases} \\ +100 \begin{cases} 64 = 0\,0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 32 = 0\,0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 4 = 0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \end{cases} \end{cases} \\ +90 = \begin{cases} +\ 80 \begin{cases} 64 = 0\,0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 16 = 0\,0\,0\,0\,0\,1\,0\,0\,0\,0 \end{cases} \\ +\ 10 \begin{cases} 8 = 0\,0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 2 = 0\,0\,0\,0\,0\,0\,0\,0\,1\,0 \end{cases} \end{cases} \end{cases} \\
\qquad +\ 9 = +\ 9 \begin{cases} 8 = 0\,0\,0\,0\,0\,0\,1\,0\,0\,0 \\ 1 = 0\,0\,0\,0\,0\,0\,0\,0\,0\,1 \end{cases} \\
\hline
\qquad\qquad\qquad 1\,1\,1\,1\,1\,0\,0\,1\,1\,1
\end{array}$$



ASSIGNMENTS

The BCD-to-binary conversion function may be implemented with PACE as a single-entry subroutine. The flowchart and program listing that follow assume that a 19-address block of memory is dedicated to storage of a binary look-up table, that AC0 and AC1 are used as input data registers for entry of the 5-digit BCD value, that all four accumulators are used as working registers during execution of the subroutine, and that the result of the conversion is stored in AC0 at the end of the subroutine.

FUNCTIONAL OPERATION

This routine uses two look-up tables to perform the BCD-to-binary conversion. The first table converts the BCD numbers in AC0, and the second table converts the BCD number in AC1. Each of the 16 bits of the BCD numbers in AC0 and the 3 bits of the BCD number in AC1 have a binary equivalent value as shown below.

AC0 BIT	BINARY VALUE (if bit set)	AC0 BIT	BINARY VALUE (if bit set)
0	1	8	100
1	2	9	200
2	4	10	400
3	8	11	800
4	10	12	1000
5	20	13	2000
6	40	14	4000
7	80	15	8000

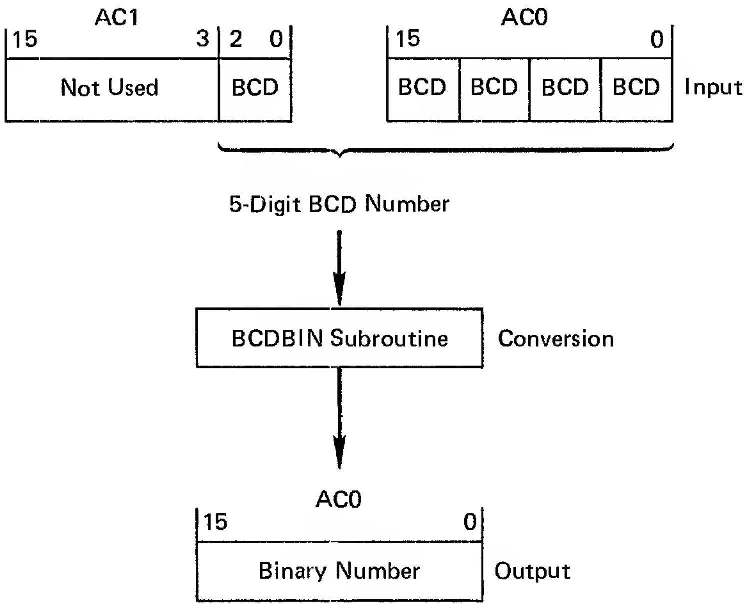
AC1 BIT	BINARY VALUE (if bit set)
0	10000
1	20000
2	40000

Each bit in AC0 and the three less-significant bits in AC1 are tested. If a bit is high, its binary value is added to a sum in AC2.

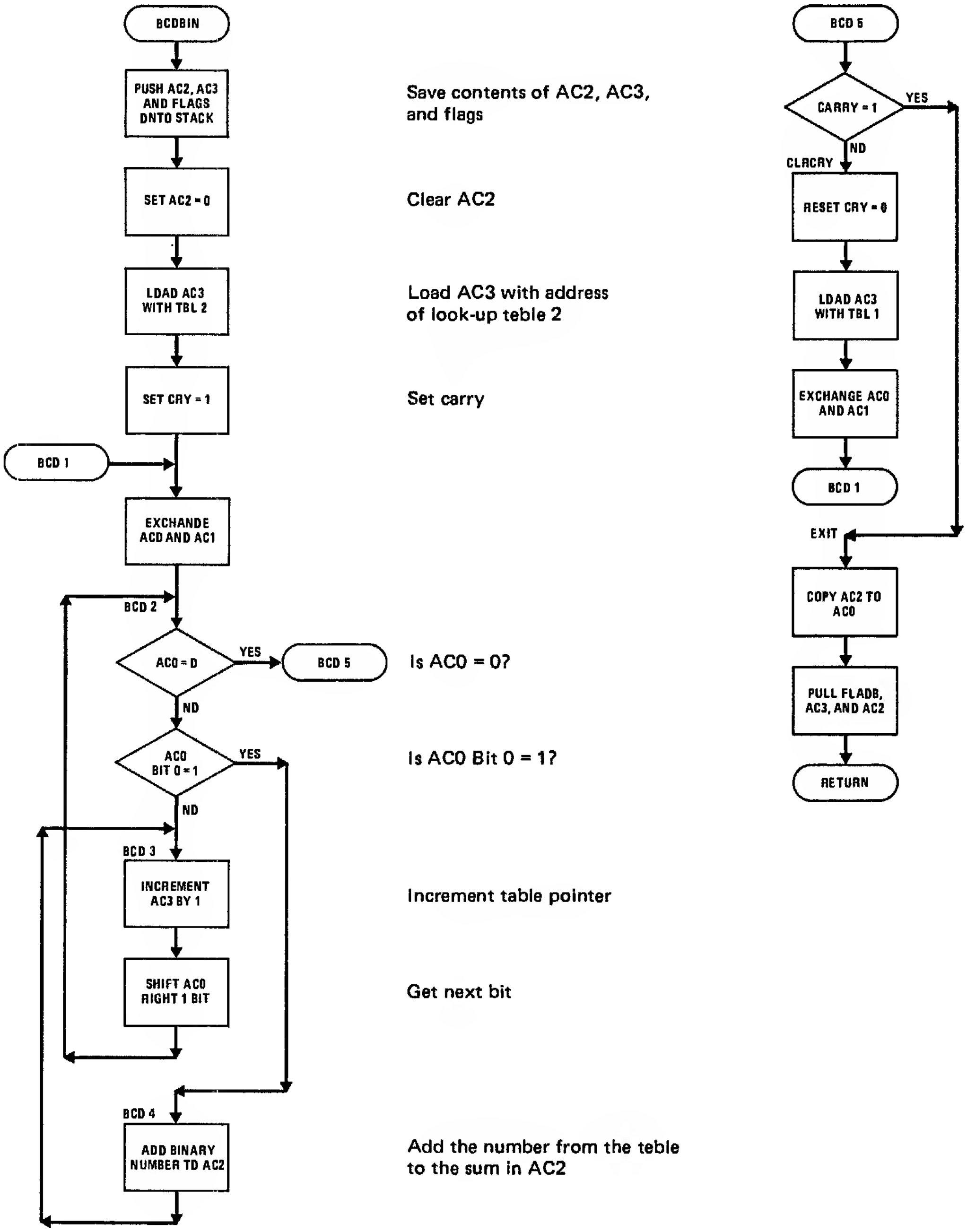
The BCD-to-Binary (BCDBIN) subroutine is entered with the BCD number to be converted in AC1 and AC0. AC1 contains the most-significant BCD digit, and AC0 contains the four less-significant digits. The routine returns the binary number in AC0. The figure below illustrates the operation of the routine.

Upon entering the BCDBIN routine, registers AC2, AC3, and the status flags are saved on the stack. AC2 is cleared, and AC3 is loaded with the address of the look-up table used to process the BCD value in AC0. The carry is set to indicate the program is processing with look-up table 2 (TBL2). The program then goes into a loop, first testing if AC0 equals zero. If it does not, bit 0 of AC0 is tested. If bit 0 equals one, the binary equivalent of the bit is added to a sum in AC2. In the next step the table pointer in AC2 is incremented by one. AC0 is shifted right one position, and the program branches to the beginning of the loop (LOOP) to process the next bit.

When AC0 equals zero, the program branches to test the carry (TESTCY). If the carry is set, it is cleared, the address of the second look-up table (TBL1) is loaded into AC3, AC0 is exchanged with AC1, and the program jumps back to the beginning of the conversion loop (LOOP). If the carry is already cleared, AC2 is copied to AC0, the flags and registers are restored, and the program returns.



FLOW CHART



Save contents of AC2, AC3, and flags

Clear AC2

Load AC3 with address of look-up teble 2

Set carry

Is AC0 = 0?

Is AC0 Bit 0 = 1?

Increment table pointer

Get next bit

Add the number from the teble to the sum in AC2

Clear carry

Loed AC3 with the address of look-up table 1

Restore flags, AC3, and AC2

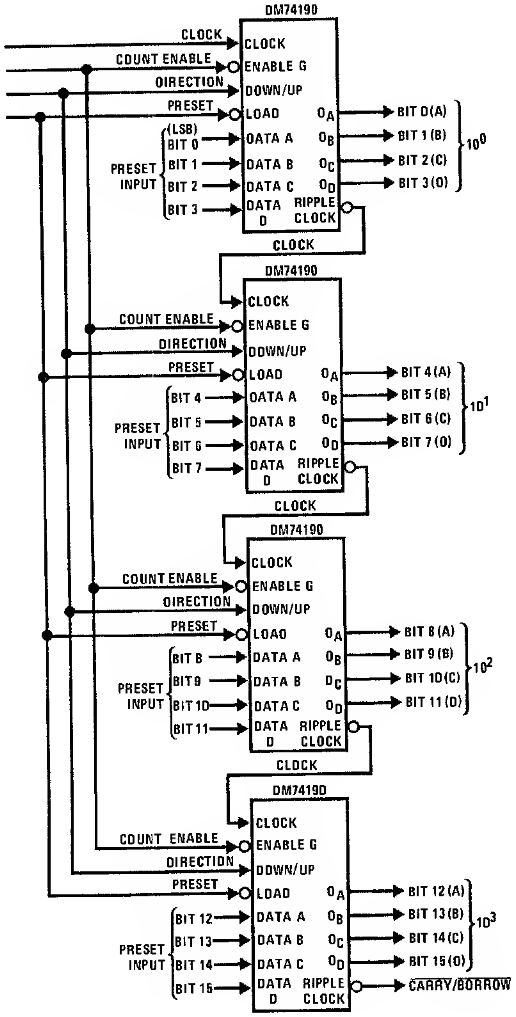
PROGRAM LISTING

```
                    ;        BCD TO BINARY
     ØØØØ     ACØ      =        Ø
     ØØØ1     AC1      =        1
     ØØØ2     AC2      =        2
     ØØØ3     AC3      =        3
     ØØØ7     CRY      =        7
     ØØØA     CARRY    =        1Ø
ØØØØ 62ØØ A   BCDBIN:  PUSH     AC2              ;SAVE AC2 ON STACK
ØØØ1 63ØØ A            PUSH     AC3              ;SAVE AC3 ON STACK
ØØØ2 ØCØØ A            PUSHF                     ;SAVE FLAGS
ØØØ3 52ØØ A            LI       AC2,Ø            ;CLEAR AC2
ØØØ4 CD24 A            LD       AC3,TBL2         ;LOAD ADDRESS OF LOOKUP
ØØØ5 378Ø A            SFLG     CRY              ;SET CARRY = 1
ØØØ6 6DØØ A   BCD1:    RXCH     ACØ,AC1          ;EXCHANGE ACØ AND AC1
ØØØ7 41Ø6 A   BCD2:    BOC      1,BCD5           ;BRANCH IF ACØ = Ø
ØØØ8 43Ø3 A            BOC      3,BCD4           ;BRANCH IF ACØ BIT Ø = 1
ØØØ9 7BØ1 A   BCD3:    AISZ     AC3,1            ;INCREMENT TABLE POINTER
ØØØA 2CØ2 A            SHR      ACØ,1,Ø          ;SHIFT ACØ RIGHT 1 BIT
ØØØB 19FB A            JMP      BCD2             ;
ØØØC EBØØ A   BCD4:    ADD      AC2,Ø(AC3)       ;ADD BINARY NUMBER
ØØØD 19FB A            JMP      BCD3             ;
ØØØE 4AØ1 A   BCD5:    BOC      CARRY,CLRCRY     ;BRANCH IF CARRY = 1
ØØØF 19Ø3 A            JMP      EXIT             ;
ØØ1Ø 37ØØ A   CLRCRY:  PFLG     CRY              ;CLEAR CARRY
ØØ11 CDØ6 A            LD       AC3,TBL1         ;LOAD ADDRESS OF LOOKUP
ØØ12 19F3 A            JMP      BCD1             ;
ØØ13 5C8Ø A   EXIT:    RCPY     AC2,ACØ          ;COPY AC2 TO ACØ
ØØ14 1ØØØ A            PULLF                     ;RESTORE FLAGS
ØØ15 67ØØ A            PULL     AC3              ;RESTORE REGISTERS
ØØ16 66ØØ A            PULL     AC2              ;
ØØ17 8ØØØ A            RTS                       ;RETURN
ØØ18 ØØ19 T   TBL1:    .WORD    .+1              ;LOOKUP TABLE 1
ØØ19 ØØØ1 A            .WORD    1
ØØ1A ØØØ2 A            .WORD    2
ØØ1B ØØØ4 A            .WORD    4
ØØ1C ØØØ8 A            .WORD    8
ØØ1D ØØØA A            .WORD    1Ø
ØØ1E ØØ14 A            .WORD    2Ø
ØØ1F ØØ28 A            .WORD    4Ø
ØØ2Ø ØØ5Ø A            .WORD    8Ø
ØØ21 ØØ64 A            .WORD    1ØØ
ØØ22 ØØC8 A            .WORD    2ØØ
ØØ23 Ø19Ø A            .WORD    4ØØ
ØØ24 Ø32Ø A            .WORD    8ØØ
ØØ25 Ø3E8 A            .WORD    1ØØØ
ØØ26 Ø7DØ A            .WORD    2ØØØ
ØØ27 ØFAØ A            .WORD    4ØØØ
ØØ28 1F4Ø A            .WORD    8ØØØ
ØØ29 ØØ2A T   TBL2:    .WORD    .+1              ;LOOKUP TABLE 2
ØØ2A 271Ø A            .WORD    1ØØØØ
ØØ2B 4E2Ø A            .WORD    2ØØØØ
ØØ2C 9C4Ø A            .WORD    4ØØØØ
     ØØØØ              .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows four DM74190 devices cascaded to form a 4-digit up/down BCD counter. For this application, counting is enabled while the Count Enable signal is low, and the direction of counting is selected by the state of the Direction signal. When the Direction signal is set low to select up-counting, each counter stage is internally configured to provide the RIPPLE CLOCK output as a look-ahead carry, and incrementing of a counter stage occurs when the previous stage is clocked from 9 to 0. Conversely, when the Direction signal is set high to select down-counting, each counter stage is internally configured to provide the RIPPLE CLOCK output as a look-ahead borrow, and decrementing of a counter stage occurs when the previous stage is clocked from 0 to 9. Thus, a RIPPLE CLOCK output is provided by the last stage when the counter is incremented to the maximum value of 9999 or decremented to the minimum value of 0000.



ASSIGNMENTS

The 4-stage up/down BCD counter function may be implemented with PACE as a multiple-entry subroutine. The flowchart and program listing that follow assume that a memory location is dedicated to storage of the count, that AC0 is used as a working register for altering the stored count, and that input/output assignments are as listed below.

INPUTS:

DM74190	PACE
Clear	CLEAR entry to Up/Down BCD Counter subroutine
Load	PRESET entry to Up/Down BCD Counter subroutine
Count Up (Count Enable, Direction, Clock)	INCREMENT entry to Up/Down BCD Counter subroutine (clock rate is equal to frequency of calling)
Count Down (Count Enable, Direction, Clock)	DECREMENT entry to Up/Down BCD Counter Subroutine (clock rate is equal to frequency of calling)

OUTPUTS:

DM74190	PACE
0000-9999	Contents of memory location COUNT
RIPPLE CLOCK (last stage)	Status Register bit 7 (carry flag)

FUNCTIONAL OPERATION

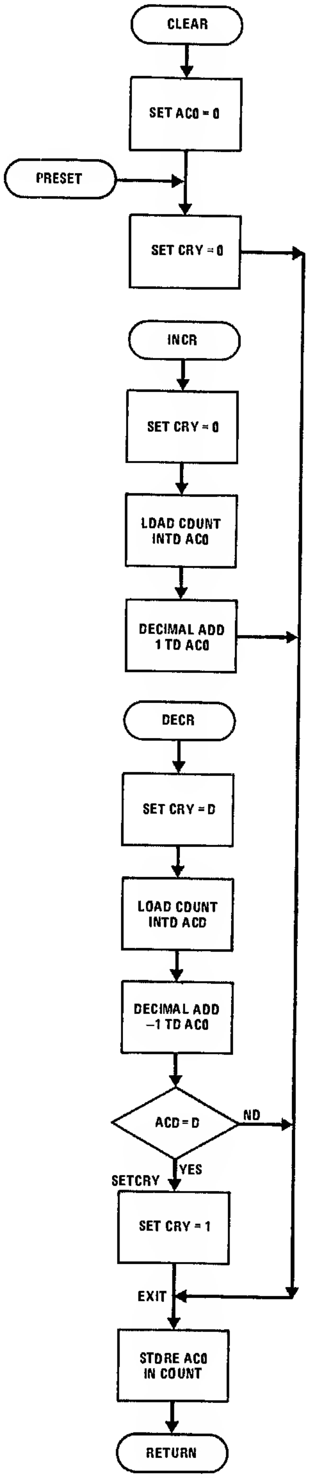
This program is written as a multiple-entry subroutine that clears, presets, increments, or decrements a 4-digit BCD counter. When the subroutine is entered at the CLEAR address, the contents of AC0 are set to zero, the carry flag is reset to clear any previous status (see the preface), and the contents of AC0 are loaded into COUNT to initialize the stored value to zero. When the subroutine is entered at the PRESET address, it is assumed that the desired preset value has already been loaded into AC0 by the main program so the contents of AC0 are not altered during execution of the subroutine. Thus, after the carry flag is reset the contents of AC0 are loaded into COUNT to initialize the stored count to some value between 0000_{10} and 9999_{10}.

The INCREMENT entry to the subroutine is functionally equivalent to configuring the DM74190 counter for up-counting. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0 after the carry flag is reset; the contents of AC0 are then incremented by one via a Decimal Add (DECA) + 1 instruction, and the new value is returned to COUNT. Use of the DECA + 1 instruction allows the stored count to be treated as a 4-digit decimal number and the carry flag to be set when AC0 is incremented from 9999_{10} to 0000_{10}.

The DECREMENT entry to the subroutine is functionally equivalent to configuring the DM74190 counter for down-counting. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0 after the carry flag is reset; the contents of AC0 are then decremented by one via a Decimal Add (DECA) −1 instruction, and the result is tested via a Branch-On-Condition (BOC) instruction. If the new value in AC0 equals zero, the carry flag is set to indicate that AC0 has been decremented to the minimum value; if the new value in AC0 is not equal to zero, the carry flag is allowed to remain reset. The new value in AC0 is then returned to COUNT and the subroutine is exited with the carry flag in the appropriate state.

Since the carry flag is set by the subroutine only when the stored count is incremented or decremented to zero, it can be tested upon return to the main program to detect completion of a normal count sequence.

FLOW CHART



Clear AC0

Clear carry

Load the previous count

Add 1 to the count

Load the previous count

Subtract 1 from the count

Is the count = 0?

Set the carry

PROGRAM LISTING

```
                     ;         UP-DOWN BCD COUNTER
          ØØØØ       ACØ       =        Ø
          ØØØ7       CRY       =        7
     ØØØØ 5ØØØ  A    CLEAR:    LI       ACØ,Ø            ;SET ACØ = Ø
     ØØØ1 37ØØ  A    PRESET:   PFLG     CRY              ;SET CARRY = Ø
     ØØØ2 19Ø8  A              JMP      EXIT             ;
     ØØØ3 C1ØB  A    INCR:     LD       ACØ,COUNT        ;LOAD COUNT INTO ACØ
     ØØØ4 37ØØ  A              PFLG     CRY              ;SET CARRY = Ø
     ØØØ5 89ØA  A              DECA     ACØ,ONE          ;DECIMAL ADD 1 TO ACØ
     ØØØ6 19Ø4  A              JMP      EXIT             ;
     ØØØ7 C1Ø7  A    DECR:     LD       ACØ,COUNT        ;LOAD COUNT INTO ACØ
     ØØØ8 37ØØ  A              PFLG     CRY              ;SET CARRY = Ø
     ØØØ9 89Ø7  A              DECA     ACØ,MINONE       ;DECIMAL ADD -1 TO ACØ
     ØØØA 41Ø2  A              BOC      1,SETCRY         ;BRANCH IF ACØ = Ø
     ØØØB D1Ø3  A    EXIT:     ST       ACØ,COUNT        ;STORE ACØ IN COUNT
     ØØØC 8ØØØ  A              RTS                       ;RETURN
     ØØØD 378Ø  A    SETCRY:   SFLG     CRY              ;SET CARRY = 1
     ØØØE 19FC  A              JMP      EXIT             ;
     ØØØF ØØØØ  A    COUNT:    .WORD    Ø                ;COUNTER SAVE
     ØØ1Ø ØØØ1  A    ONE:      .WORD    1                ;CONSTANT
     ØØ11 9999  A    MINONE:   .WORD    Ø9999            ;1Ø'S COMPLEMENT -1
          ØØØØ                 .END
```

MSI

DM54/DM74190,LS190,191,LS191

Synchronous Up/Down Counters with Mode Control

General Description

These circuits are synchronous, reversible, up/down counters. The 191 and LS191 are 4-bit binary counters and the 190 and LS190 are BCD counters. Synchronous operation is provided by having all flip-flops clocked simultaneously, so that the outputs change simultaneously when so instructed by the steering logic. This mode of operation eliminates the output counting spikes normally associated with asynchronous (ripple clock) counters.

The outputs of the four master-slave flip-flops are triggered on a low-to-high level transition of the clock input, if the enable input is low. A high at the enable input inhibits counting. Level changes at either the enable input or the down/up input should be made only when the clock input is high. The direction of the count is determined by the level of the down/up input. When low, the counter counts up and when high, it counts down.

These counters are fully programmable; that is, the outputs may be preset to either level by placing a low on the load input and entering the desired data at the data inputs. The output will change independent of the level of the clock input. This feature allows the counters to be used as modulo-N dividers by simply modifying the count length with the preset inputs.

The clock, down/up, and load inputs are buffered to lower the drive requirement; which significantly reduces the number of clock drivers, etc., required for long parallel words.

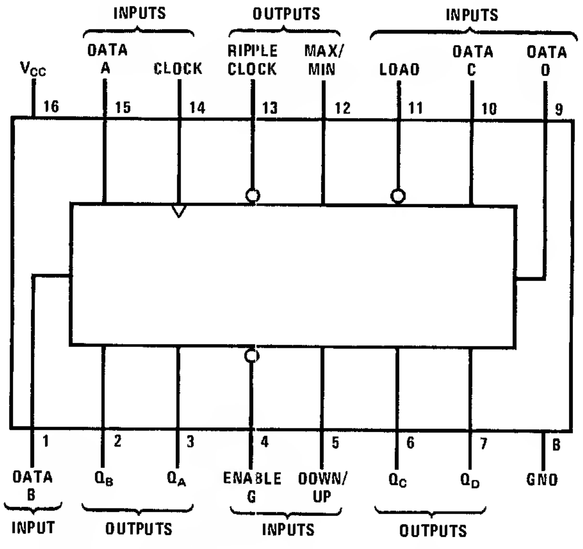
Two outputs have been made available to perform the cascading function: ripple clock and maximum/minimum count. The latter output produces a high-level output pulse with a duration approximately equal to one complete cycle of the clock when the counter overflows or underflows. The ripple clock output produces a low-level output pulse equal in width to the low-level portion of the clock input when an overflow or underflow condition exists. The counters can be easily cascaded by feeding the ripple clock output to the enable input of the succeeding counter if parallel clocking is used, or to the clock input if parallel enabling is used. The maximum/minimum count output can be used to accomplish look-ahead for high-speed operation.

Features

- Counts 8-4-2-1 BCD or binary
- Single down/up count control line
- Count enable control input
- Ripple clock output for cascading
- Asynchronously presettable with load control
- Parallel outputs
- Cascadable for n-bit applications

TYPE	AVERAGE PROPAGATION DELAY	TYPICAL CLOCK FREQUENCY	TYPICAL POWER DISSIPATION
190, 191	20 ns	25 MHz	325 mW
LS190, LS191	20 ns	25 MHz	100 mW

Connection Diagram



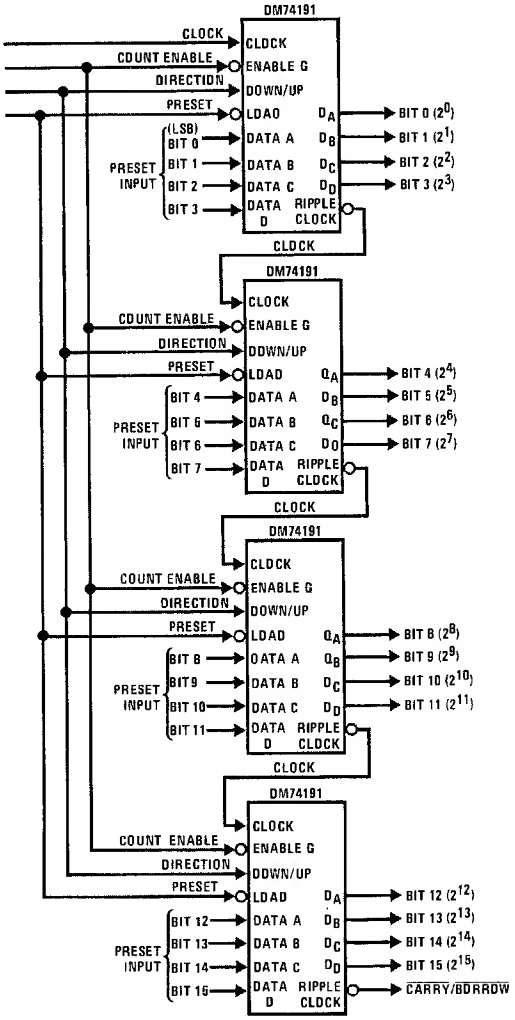
Asynchronous inputs: Low input to load sets $Q_A = A$, $Q_B = B$, $Q_C = C$, and $Q_D = D$

54190/74190(J), (N), (W); 54LS190/74LS190(J), (N), (W); 54191/74191(J), (N), (W); 54LS191/74LS191(J), (N), (W)

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows how four DM74191 devices are cascaded to form a 16-bit up/down binary counter. For this application, counting is enabled while the Count Enable signal is low, and the direction of counting is selected by the state of the Direction signal. When the Direction signal is set low to select up-counting, each counter stage is internally configured to provide the RIPPLE CLOCK output as a look-ahead carry, and incrementing of a counter stage occurs when the previous stage is clocked from 15 to 0. Conversely, when the Direction signal is set high to select down-counting, each counter stage is internally configured to provide the RIPPLE CLOCK output as a look-ahead borrow, and decrementing of a counter stage occurs when the previous stage is clocked from 0 to 15. Thus, a RIPPLE CLOCK output is provided by the last stage when the counter is incremented to the maximum value of FFFF or decremented to the minimum value of 0000.



ASSIGNMENTS

The 16-bit up/down binary counter function may be implemented with PACE by a multiple-entry subroutine. The flowchart and program listing that follow assume that a memory location is dedicated to storage of the count, that AC0 is used as a working register for altering the stored count, and that input/output assignments are as listed below.

INPUTS:

DM74190	PACE
Clear	CLEAR entry to Up/Down Binary Counter subroutine
Load	PRESET entry to Up/Down Binary Counter subroutine
Count Up (Count Enable, Direction, Clock)	INCREMENT entry to Up/Down Binary Counter subroutine (clock rate is equal to frequency of calling)
Count Down (Count Enable, Direction, Clock)	DECREMENT entry to Up/Down Binary Counter subroutine (clock rate is equal to frequency of calling)

OUTPUTS:

DM74190	PACE
0000-9999	Contents of memory location COUNT
RIPPLE CLOCK (last stage)	Status Register bit 7 (carry flag)

FUNCTIONAL OPERATION

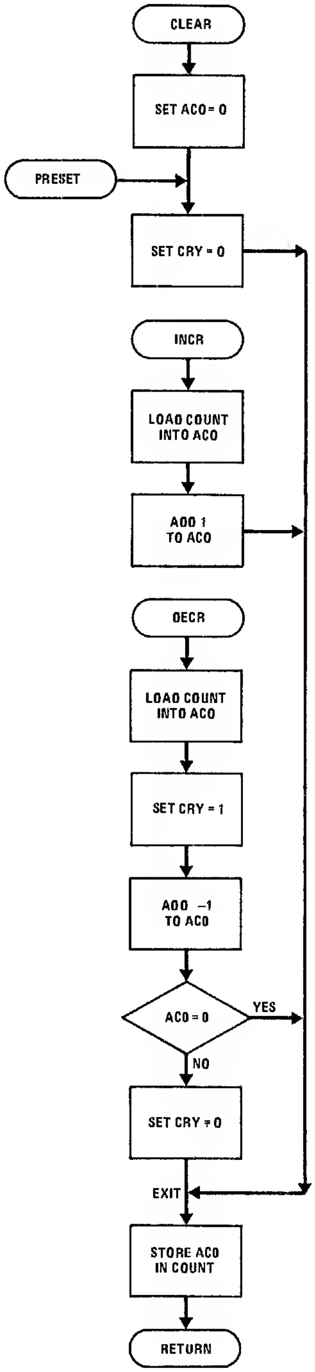
This program is written as a multiple-entry subroutine that clears, presets, increments, or decrements a binary counter. When the subroutine is entered at the CLEAR address, the contents of AC0 are set to zero, the carry flag is reset to clear any previous status (see the preface), and the contents of AC0 are loaded into memory-location COUNT to initialize the stored value to zero. When the subroutine is entered at the PRESET address, it is assumed that the desired preset value has already been loaded into AC0 by the main program so the contents of AC0 are not altered during execution of the subroutine. Thus, after the carry flag is reset, the contents of AC0 are loaded into COUNT to initialize the stored count to some value between 0000 and FFFF.

The INCREMENT entry to the subroutine is functionally equivalent to configuring the DM74191 counter for up-counting. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0; the contents of AC0 are then incremented by one via an ADD + 1 instruction, and the new value is returned to COUNT. Use of the ADD + 1 instruction enables the stored count to be treated as a 16-bit number and the carry flag to be set when AC0 is incremented from FFFF to 0000.

The DECREMENT entry to the subroutine is functionally equivalent to configuring the DM74191 counter for down-counting. When the subroutine is entered at this address, the value stored in COUNT is loaded into AC0, the carry flag is set high, and the contents of AC0 are decremented by one via an AISZ −1 instruction. Use of the AISZ −1 instruction automatically tests the result for zero but does not affect the state of the carry flag. If the result is not zero, a PFLG CRY instruction is executed to reset the carry flag; the new value in AC0 is then returned to COUNT to complete the subroutine. If the result is zero, the PFLG CRY instruction is skipped and the subroutine is exited with the carry flag set after the new value in AC0 is returned to COUNT.

Since the carry flag is set by the subroutine only when the stored count is incremented or decremented to zero, it can be tested upon return to the main program to detect completion of a normal count sequence.

FLOW CHART



Clear AC0

Clear carry

Load the previous count

Add 1 to the count

Load the previous count

Set the carry

Subtract 1 from the count

Is the count = 0?

Clear the carry

Save the count

PROGRAM LISTING

```
                   ;        UP-DOWN BINARY COUNTER
     ØØØØ          ACØ      =        Ø
     ØØØ7          CRY      =        7
ØØØØ 5ØØØ  A       CLEAR:   LI       ACØ,Ø           ;SET ACØ = Ø
ØØØ1 37ØØ  A       PRESET:  PFLG     CRY             ;RET CARRY = Ø
ØØØ2 19Ø7  A                JMP      EXIT            ;
ØØØ3 C1Ø8  A       INCR:    LD       ACØ,COUNT       ;LOAD COUNT INTO ACØ
ØØØ4 E1Ø8  A                ADD      ACØ,ONE         ;ADD 1 TO ACØ
ØØØ5 19Ø4  A                JMP      EXIT            ;
ØØØ6 C1Ø5  A       DECR:    LD       ACØ,COUNT       ;LOAD COUNT INTO ACØ
ØØØ7 378Ø  A                SFLG     CRY             ;SET CARRY = 1
ØØØ8 78FF  A                AISZ     ACØ,-1          ;ADD -1 TO ACØ SKIP IF Ø
ØØØ9 37ØØ  A                PFLG     CRY             ;SET CARRY = Ø
ØØØA D1Ø1  A       EXIT:    ST       ACØ,COUNT       ;STORE ACØ IN COUNT
ØØØB 8ØØØ  A                RTS                      ;RETURN
ØØØC ØØØØ  A       COUNT:   .WORD    Ø               ;COUNTER SAVE
ØØØD ØØØ1  A       ONE:     .WORD    1               ;CONSTANT
     ØØØØ                   .END
```

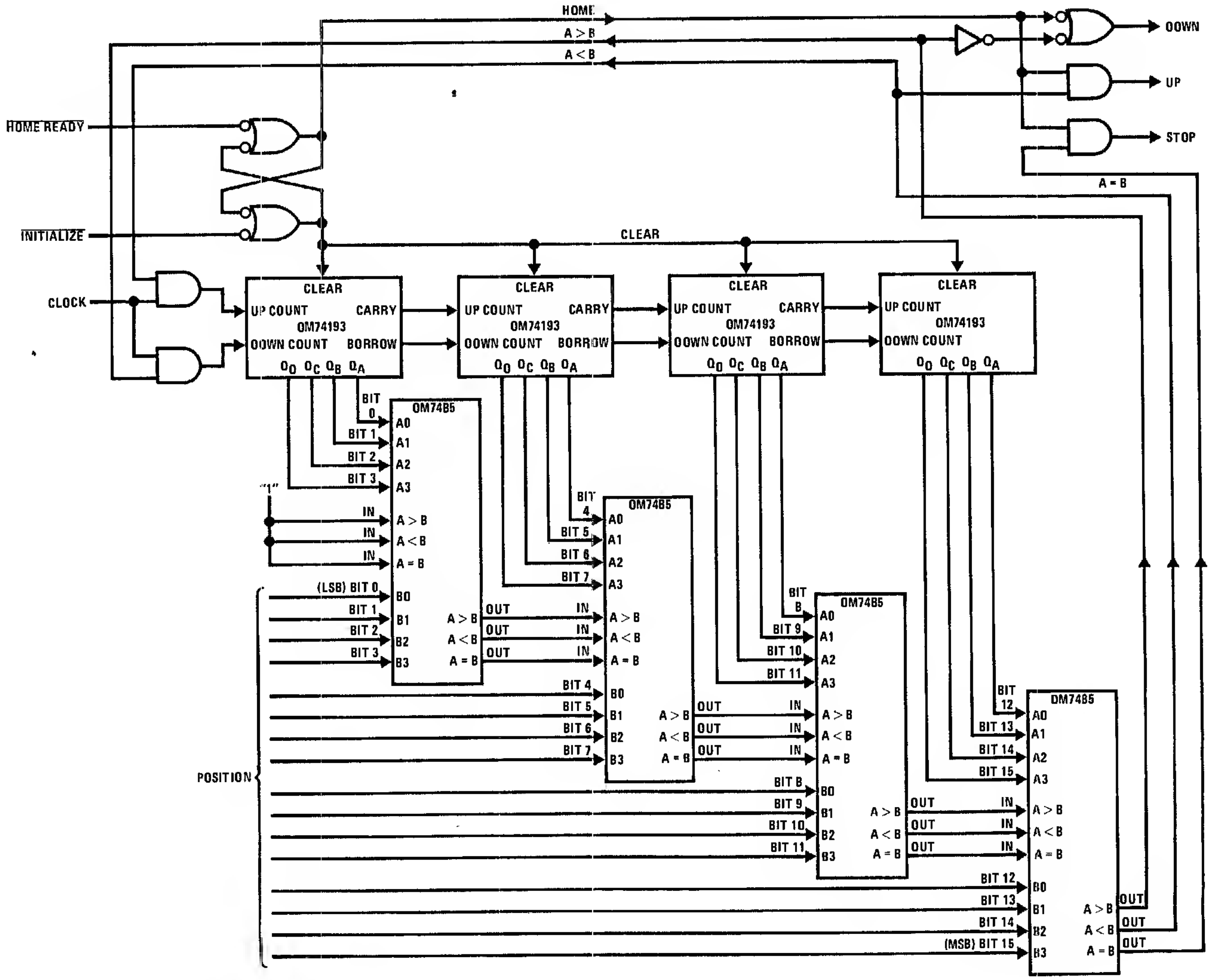
Chapter 4
THE SIMULATIONS

Part 2: SUBSYSTEMS

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows a 16-bit up/down binary counter interconnected with a 16-bit magnitude comparator to form the control logic for a digitally-controlled servo. (Operation of the counter and comparator is covered on pp. 4-53 to 4-56 and 4-28 to 4-30). For this application, the control logic is implemented for position control. When power is first turned on, the low-going Initialize pulse sets the Input flip-flop to hold the counters reset and the A > B output high, thereby causing the external servo element to be driven to the zero reference position. When the external servo element reaches the zero reference position, the Input flip-flop is reset, and the low-going Home Ready pulse and normal operation of the servo are enabled over the complete range of 0000 through FFFF. (It is assumed that the external servo element provides one clock pulse for each increment of motion in either the up or down direction.) With normal operation enabled, the A > B, A < B, and A = B outputs serve to drive the servo element to the position indicated by the 16-bit Position input. If, for example, the Position input is a greater value than the output of the counter, the A < B output causes the servo to be driven in the up direction and the resultant clock input is applied as an up clock to the counter; when the counter is subsequently counted-up to the Position value, the A < B output goes low and the A = B output goes high to stop servo motion. The servo then holds its current position until the value of the Position input is increased or decreased to move the servo element up or down, respectively.



ASSIGNMENTS

The digital servo function may be implemented with PACE by a single-entry subroutine. The flowchart and program listing that follow assume that memory locations are dedicated to storage of the current and desired servo positions, that accumulator AC1 is used as an input data register for entry of the desired servo position and also as a working register (along with accumulator AC0) to determine when the servo is at the desired position, and that input/output signal assignments are as listed below.

HARDWARE CONFIGURATION	PACE
Clock	JC15
A > B	Flag 13 set (drive servo down)
A < B	Flag 14 set (drive servo up)
A = B	Flag 13 and 14 reset (stop servo)

FUNCTIONAL DESCRIPTION

Two versions of the digital servo subroutine are provided. The first version uses the 16-bit Comparator routine (COMP16), described on pp. 4-28 to 4-30, to illustrate a building-block approach to subroutine generation. The second version performs the comparison within the servo subroutine; this version serves to show some of the options available to the programmer in any given application. The assignments specified above are valid for both versions of the servo subroutine.

Upon entry to the first digital servo subroutine (SERVO1), the contents of AC0 are saved on the stack (so that they can be restored at the end of the subroutine) and the contents of AC1 are loaded into memory-location NEW (which frees AC1 for use as a working register). The contents of OLD and NEW are then loaded into AC0 and AC1, respectively, and the COMP16 subroutine is called to compare the two values. When the COMP16 subroutine is completed, the results of the comparison will be stored in AC0 as follows:

- Bit 0 of AC0 will be high if the two values were equal.
- Bit 1 of AC0 will be high if the value in AC0 was greater than the value in AC1.
- Bit 2 of AC0 will be high if the value in AC0 was less than the value in AC1.

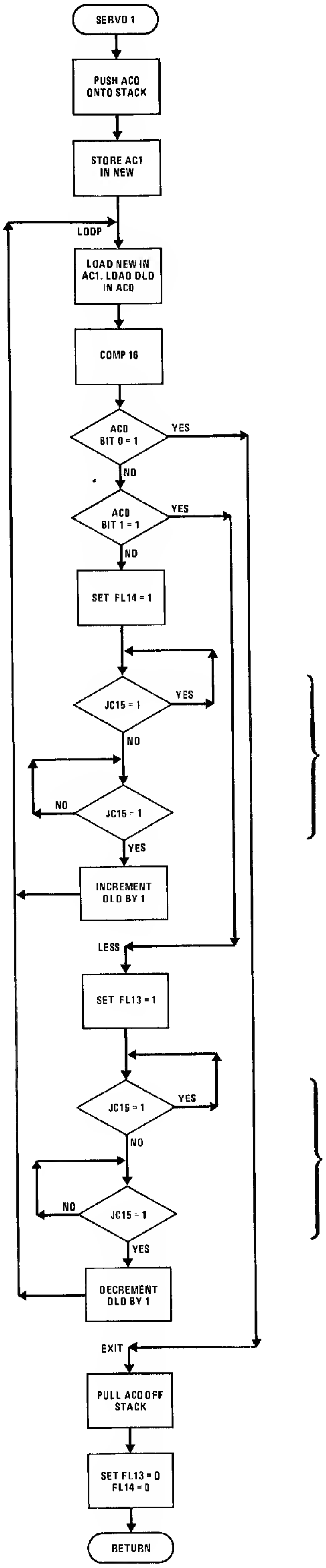
After the results of the comparison are stored in AC0, bit 0 of AC0 is tested for the high state to determine whether the two values were equal.

If bit 0 is high, the servo is at the desired position, and the subroutine is exited after the original contents of AC0 are restored from the stack; flags 13 and 14 are pulsed reset to ensure that servo motion is inhibited.

If bit 0 of AC0 is low, bit 1 is tested to determine whether the servo needs to be driven up or down. Flag 13 or 14 is then set to enable downward or upward motion, respectively, and the JC15 input is tested to detect the positive-going edge of the resultant clock input. Upon detection of the positive-going clock edge, the contents of OLD are incremented or decremented as appropriate, and the subroutine returns to the LOOP address. The "compare and count loop" is then repeated continuously until the servo arrives at the desired position (i.e., the contents of OLD are the same as the contents of NEW). When this occurs, bit 0 of AC0 will be high following the return from the COMP16 subroutine, and the compare and count loop will be terminated by the resultant branch to the EXIT address. The original contents of AC0 will then be restored from the stack, flags 13 and 14 will be reset to terminate servo motion, and a return to the main program will be effected via an RTS instruction.

The second digital servo subroutine (SERVO2) is similar to the first except for the comparison function, which is now performed within the subroutine. This is accomplished by twos-complementing AC1 after AC0 is stored on the stack, then loading OLD into AC0 and adding the contents of AC0 and AC1 together. In effect, this is a standard binary subtraction, one which does not alter the contents of AC1. If the result of the subtraction (stored in AC0) is zero, it indicates that the servo is at the desired position. Similarly if the result is not zero, the state of the carry flag indicates whether the servo needs to be driven up (carry flag reset because AC0 < AC1) or down (carry flag set because AC0 > AC1). Thus, after the subtraction is performed, the SERVO2 subroutine tests AC0 for zero and/or the state of the carry flag to control servo motion in the same manner as described above for the SERVO1 subroutine.

FLOW CHART, SERVO 1



Save contents of AC0 on stack

Save new count

Load old and new counts

Compare OLD and NEW

Exit if they are equal

New count < old count?

Set drive-up flag

Wait for clock transition

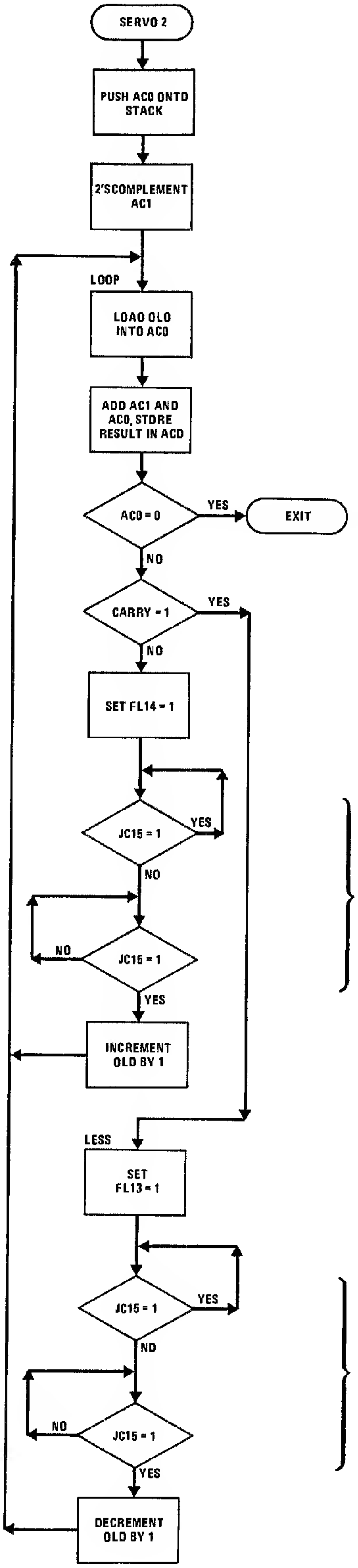
Set drive-down flag

Wait for clock transition

Restore AC0

Clear up-and-down drive control

FLOW CHART, SERVO 2



Save contents of AC0 on stack

Complement the new count in AC1
Load the old count into AC0

Add old count and complemented new count

Exit if OLD and NEW are equal

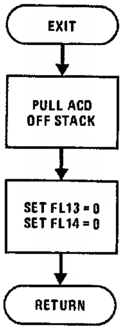
Is new count < old count?

Set drive-up flag

Wait for clock transition

Set drive-down flag

Wait for clock transition



Restore AC0

Clear up-and-down drive control

PROGRAM LISTING, SERVO 1

```
                ;        DIGITAL SERVO
      ØØØØ      ACØ      =       Ø
      ØØØ1      AC1      =       1
      ØØØA      CRY      =       1Ø            ;CARRY
      ØØØD      FL13     =       13            ;DRIVE DOWN FLAG
      ØØØE      FL14     =       14            ;DRIVE UP FLAG
      ØØØF      JC15     =       15            ;CLOCK
                         .GLOBL  COMP16        ;16 BIT COMPARATOR
ØØØØ  6ØØØ  A   SERVO:   PUSH    ACØ           ;SAVE ACØ ON STACK
ØØØ1  D515  A            ST      AC1,NEW       ;SAVE NEW COUNT
ØØØ2  C514  A   LOOP:    LD      AC1,NEW       ;LOAD AC1 WITH NEW
ØØØ3  C114  A            LD      ACØ,OLD       ;LOAD ACØ WITH OLD
ØØØ4  14Ø1  X            JSR     COMP16        ;COMPARE NEW AND OLD
ØØØ5  43ØD  A            BOC     3,EXIT        ;EXIT IF EQUAL
ØØØ6  44Ø6  A            BOC     4,LESS        ;BRANCH IF NEW < OLD
                ;        NEW COUNT GREATER THAN OLD COUNT
ØØØ7  3E8Ø  A            SFLG    FL14          ;SET DRIVE UP FLAG
ØØØ8  4FFF  A            BOC     JC15,.+Ø      ;WAIT FOR CLOCK TO GO LO
ØØØ9  4FØ1  A            BOC     JC15,.+2      ;WAIT FOR CLOCK TO GO HI
ØØØA  19FE  A            JMP     .-1           ;
ØØØB  8DØC  A            ISZ     OLD           ;INCREMENT OLD BY 1
ØØØC  19F5  A            JMP     LOOP          ;CONTINUE ASSUME NO SKIP
                ;        NEW COUNT LESS THAN OLD COUNT
ØØØD  3D8Ø  A   LESS:    SFLG    FL13          ;SET DRIVE DOWN FLAG
ØØØE  4FFF  A            BOC     JC15,.+Ø      ;WAIT FOR CLOCK TO GO LO
ØØØF  4FØ1  A            BOC     JC15,.+2      ;WAIT FOR CLOCK TO GO HI
ØØ1Ø  19FE  A            JMP     .-1           ;
ØØ11  ADØ6  A            DSZ     OLD           ;DECREMENT OLD BY 1
ØØ12  19EF  A            JMP     LOOP          ;CONTINUE IF OLD NOT Ø
ØØ13  64ØØ  A   EXIT:    PULL    ACØ           ;RESTORE ACØ
ØØ14  3DØØ  A            PFLG    FL13          ;CLEAR DRIVE DOWN FLAG
ØØ15  3EØØ  A            PFLG    FL14          ;CLEAR DRIVE UP FLAG
ØØ16  8ØØØ  A            RTS                   ;RETURN
ØØ17  ØØØØ  A   NEW:     .WORD   Ø             ;NEW COUNT
ØØ18  ØØØØ  A   OLD:     .WORD   Ø             ;OLD COUNT
      ØØØØ               .END
```

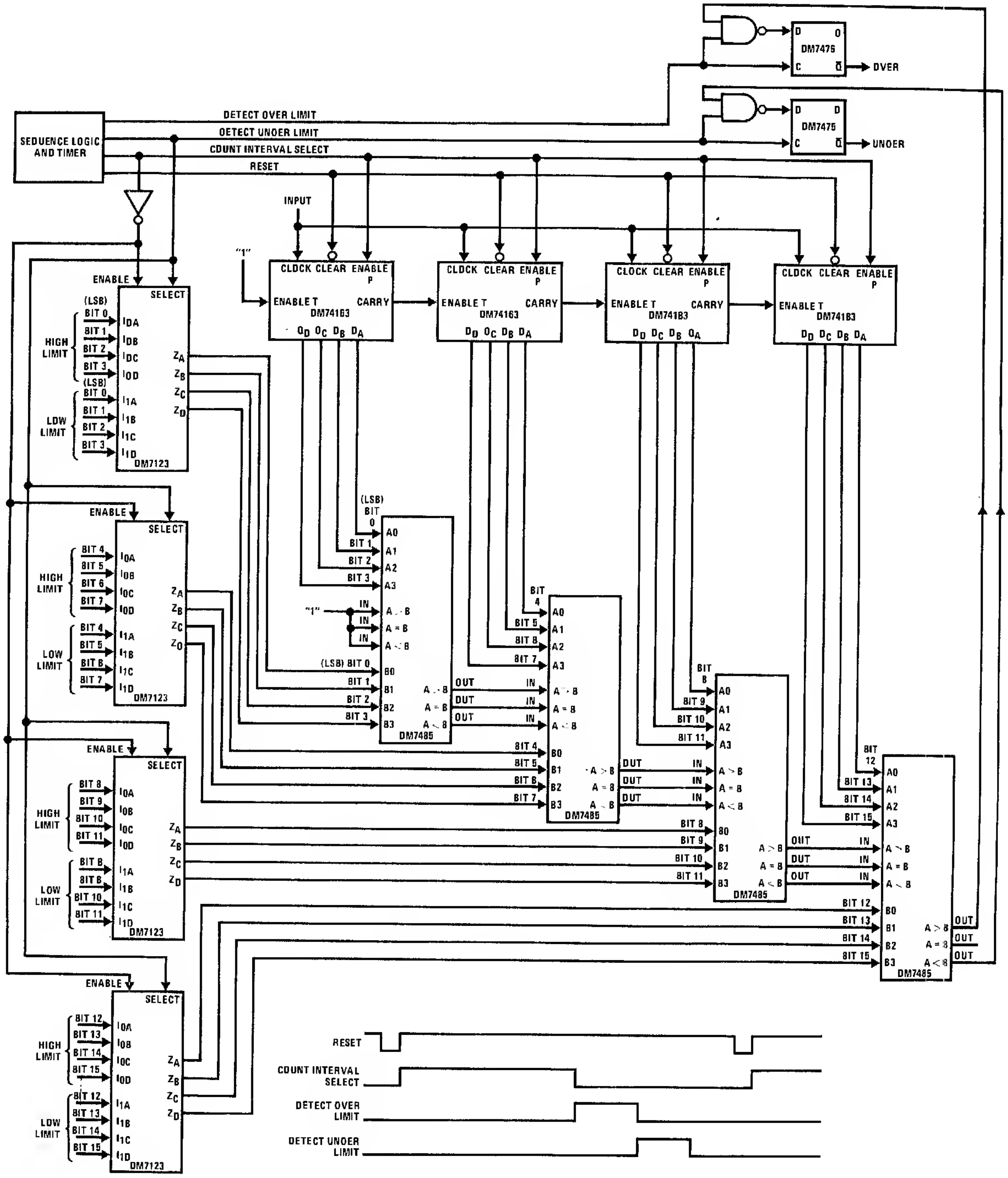
PROGRAM LISTING, SERVO 2

```
                ;        DIGITAL SERVO
      ØØØØ      ACØ      =       Ø
      ØØØ1      AC1      =       1
      ØØØA      CRY      =       1Ø            ;CARRY
      ØØØD      FL13     =       13            ;DRIVE DOWN FLAG
      ØØØE      FL14     =       14            ;DRIVE UP FLAG
      ØØØF      JC15     =       15            ;CLOCK
ØØØØ  6ØØØ  A   SERVO:   PUSH    ACØ           ;SAVE ACØ ON STACK
ØØØ1  71Ø1  A            CAI     AC1,1         ;2S COMPLEMENT NEW COUNT
ØØØ2  C113  A   LOOP:    LD      ACØ,OLD       ;LOAD OLD COUNT INTO ACØ
ØØØ3  684Ø  A            RADD    AC1,ACØ       ;(ACØ) - (AC1) -> (ACØ)
ØØØ4  41ØD  A            BOC     1,EXIT        ;EXIT IF NEW = OLD
ØØØ5  4AØ6  A            BOC     CRY,LESS      ;BRANCH IF NEW < OLD
                ;        NEW COUNT GREATER THAN OLD COUNT
ØØØ6  3E8Ø  A            SFLG    FL14          ;SET DRIVE UP FLAG
ØØØ7  4FFF  A            BOC     JC15,.+Ø      ;WAIT FOR CLOCK TO GO LO
ØØØ8  4FØ1  A            BOC     JC15,.+2      ;WAIT FOR CLOCK TO GO HI
ØØØ9  19FE  A            JMP     .-1           ;
ØØØA  8DØB  A            ISZ     OLD           ;INCREMENT OLD BY 1
ØØØB  19F6  A            JMP     LOOP          ;CONTINUE ASSUME NO SKIP
                ;        NEW COUNT LESS THAN OLD COUNT
ØØØC  3D8Ø  A   LESS:    SFLG    FL13          ;SET DRIVE DOWN FLAG
ØØØD  4FFF  A            BOC     JC15,.+Ø      ;WAIT FOR CLOCK TO GO LO
ØØØE  4FØ1  A            BOC     JC15,.+2      ;WAIT FOR CLOCK TO GO HI
ØØØF  19FE  A            JMP     .-1           ;
ØØ1Ø  ADØ5  A            DSZ     OLD           ;DECREMENT OLD BY 1
ØØ11  19FØ  A            JMP     LOOP          ;CONTINUE IF OLD NOT Ø
ØØ12  64ØØ  A   EXIT:    PULL    ACØ           ;RESTORE ACØ
ØØ13  3DØØ  A            PFLG    FL13          ;CLEAR DRIVE DOWN FLAG
ØØ14  3EØØ  A            PFLG    FL14          ;CLEAR DRIVE UP FLAG
ØØ15  8ØØØ  A            RTS                   ;RETURN
ØØ16  ØØØØ  A   OLD:     .WORD   Ø             ;OLD COUNT
      ØØØØ               .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows a 16-bit binary counter interconnected with a 16-bit magnitude comparator and a sequence-logic-and-timer circuit to form a digital tachometer. (Operation of the counter and comparator is covered on pp. 4-34 to 4-36 and 4-28 to 4-30.) For this application, the counter is enabled to count for a fixed interval by the Count Interval Select output of the sequence-logic-and-timer circuit, then the resultant output of the counter is compared with the high- and low-limit reference inputs to indicate whether the input was over, under, or within the range selected.



ASSIGNMENTS

The digital tachometer function may be implemented with PACE as a single-entry subroutine. The flowchart and program listing that follow assume that the PACE Level 2 Interrupt input is continuously driven by a low-going 10μs clock pulse at a 60 Hz rate, that AC0 is used as a working register for selecting the count interval time and detecting completion of the counting sequence, that AC1 is used as a working register for counting the number of input pulses received while counting is enabled, and that input/output assignments are as listed below.

NOTE: The Level 2 and Level 3 Interrupt clock parameters can be easily derived using either a one-shot multivibrator or an edge detector. For a detailed description of PACE interrupt signal requirements, refer to the material on PACE's interrupt system, which begins on page 3-2.

INPUTS:

DIGITAL TACHOMETER	PACE
Count Interval Select	TIMER constant (60_{10}) entered into AC0 when subroutine is called by main program, assuming that Level 2 Interrupt input is continuously driven by 10μs low-going clock pulse at 60 Hz rate.
Input	Level 3 Interrupt input driven by 10μs low-going clock pulse (maximum clock frequency is 10 kHz)
High Limit	Contents of memory location MAX
Low Limit	Contents of memory location MIN
Reset	Automatic upon completion of subroutine

OUTPUTS:

DIGITAL TACHOMETER	PACE
Over Limit	RETURN exit from subroutine
Under Limit	RETURN + 1 exit from subroutine
Within Limit	RETURN + 2 exit from subroutine

FUNCTIONAL OPERATION

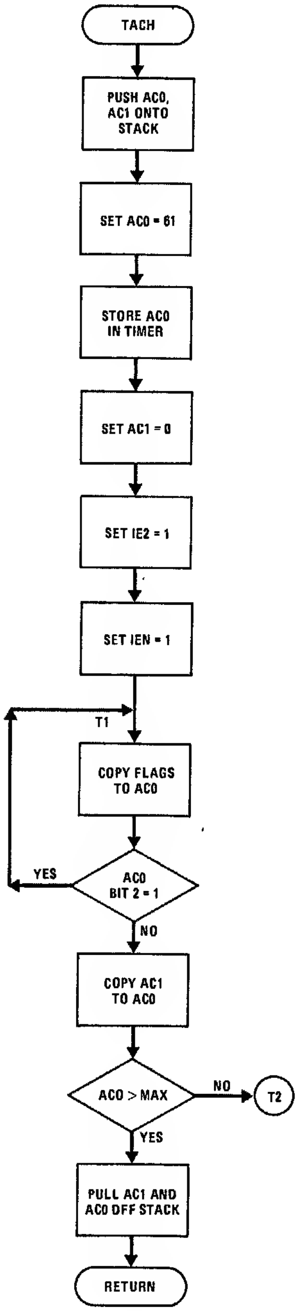
This program is written as a single-entry subroutine that enables the Level 3 Interrupt clock to be counted for a 1-second interval, and the result of the count to be compared with predetermined minimum and maximum limits. Since the subroutine requires that AC0 and AC1 be used as working registers, the first operation of the subroutine is to push AC0 and AC1 onto the stack so that their original contents can be restored at the end of the subroutine. After AC0 and AC1 are saved on the stack, decimal value 61 is loaded into memory location TIMER via AC0, and AC1 in initialized to zero. Then the Interrupt Enable 2 and master interrupt enable flags are set to enable processing of the 60 Hz, Level 2 Interrupt clock input.

After the Level 2 and master interrupt enable flags are set, a "copy status register into AC0/test AC0 bit 2" (Interrupt 2 enable flag) loop is continually executed until the first Level 2 Interrupt clock is received. Upon receipt of this input,PACE automatically branches to the Level 2 Interrupt service routine causing the contents of memory location TIMER to be decremented to 60 and the Level 3 Interrupt enable flag to be set to allow counting of the Level 3 Interrupt clock. The Return from Interrupt (RTI) instruction then causes a return to the TACH subroutine "copy register into AC0/test bit 2" loop. Since the Interrupt 2 enable flag was returned true at the start of the Level 2 Interrupt service routine, the TACH subroutine loop will be maintained until a subsequent Level 2 or Level 3 Interrupt clock is received.

After the Level 2 Interrupt service routine is executed for the first time, subsequent Level 2 Interrupt clocks will cause the contents of memory location TIMER to be decremented from 60 to zero at a 60 Hz rate to enable counting of the Level 3 Interrupt clock input for a 1-second interval. While the contents of memory location TIMER are greater than zero, the exits from the Level 2 Interrupt service routine occur with the Level 2 and Level 3 Interrupt enable flags set, which reinstate the TACH subroutine "copy flags into AC0/test AC0 bit 2 loop." Thus, each Level 3 Interrupt clock input will cause execution of the Level 3 Interrupt service routine to allow incrementing of AC1 by one and a return to the TACH subroutine "copy flags into AC0/test AC0 bit 2" loop.

When the contents of memory location TIMER are decremented to zero at the end of the 1 second counting interval, the return from the Level 2 Interrupt service routine occurs with both the Level 2 and Level 3 Interrupt flags reset to disable counting. Thus, the "copy flags into AC0/test AC0 bit 2" loop is terminated upon return to the TACH subroutine. The Level 3 Interrupt clock count stored in AC1 is then loaded into AC0 and AC0 is compared with the maximum limit stored in memory-location MAX. If the contents of AC0 are greater than the maximum limit, AC0 and AC1 are pulled from the stack to reinstate the original contents, and the subroutine is exited via a Return (RTS) instruction to provide an over-limit indication to the main program. If the contents of AC0 are less than the maximum limit, the contents of memory location MIN are subtracted from AC0 to provide an under-limit (RTS + 1) or within-limit (RTS + 2) return to the main program (after AC0 and AC1 are pulled from the stack to reinstate their original contents).

FLOW CHART



Save contents of AC0 and AC1 on stack

Load AC0 with timer constant

Store constant into timer

Clear AC1

Enable interrupt 2

Enable interrupts

Put flag into AC0

Is interrupt service routine timed out?

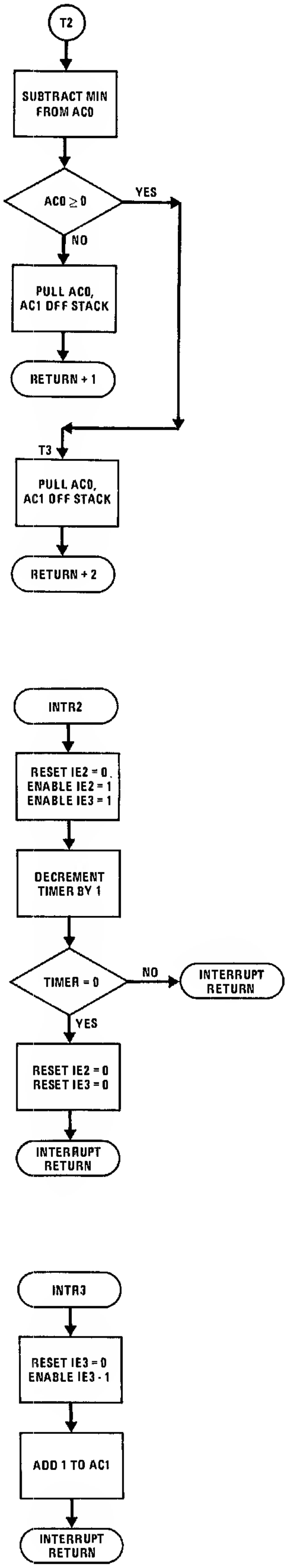
Put pulse count into AC0

Is count greater than maximum limits?

Restore AC1 and AC0

Count Over return

FLOW CHART (Continued)



(AC0) ← (AC0) – (Min)

Is count less than minimum limits?

Restore AC0 and AC1

Count Under return

Restore AC0 and AC1

Count OK return

Interrupt 2 service routine

Reset and enable interrupt 2 and enable interrupt 3

Decrement 1 second timer

Is timer zero?

Reset interrupts 2 and 3

Interrupt 3 service routine

Reset and enable interrupt 3

Add 1 to pulse count

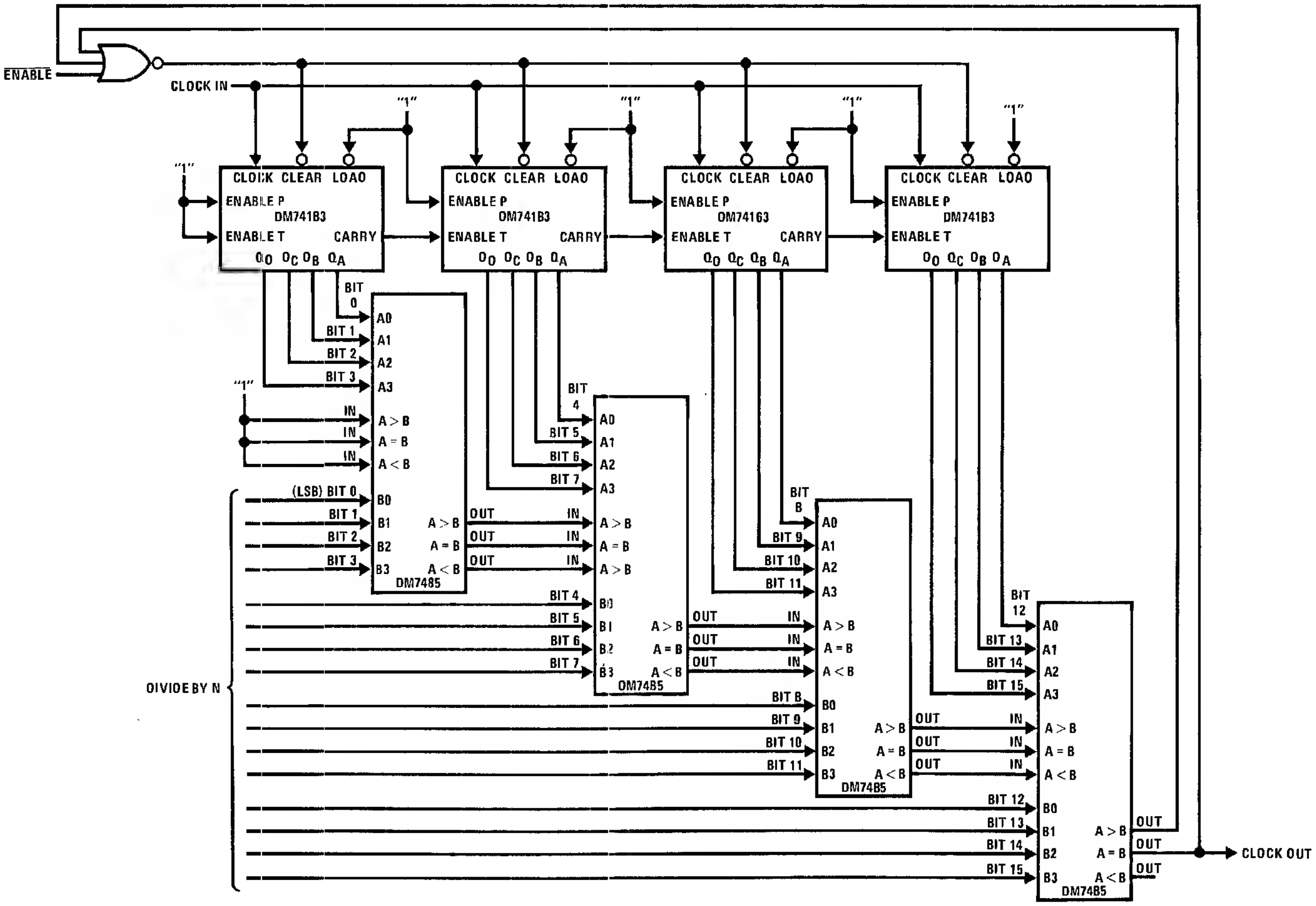
PROGRAM LISTING

```
                ;         DIGITAL TACHOMETER
      ØØØØ      ACØ       =       Ø
      ØØØ1      AC1       =       1
      ØØØ2      IE2       =       2               ;INTERRUPT 2
      ØØØ3      IE3       =       3               ;INTERRUPT 3
      ØØØ9      IEN       =       9               ;INTERRUPT ENABLE
      ØØ41      MIN       =       Ø41             ;MINIMUM LIMIT ADDRESS
      ØØ42      MAX       =       Ø42             ;MAXIMUM LIMIT ADDRESS
ØØØØ  61ØØ  A   TACH:     PUSH    AC1             ;SAVE REGISTERS ON STACK
ØØØ1  6ØØØ  A             PUSH    ACØ             ;
ØØØ2  5Ø3D  A             LI      ACØ,61          ;SET ACØ = 61 (DECIMAL)
ØØØ3  D11F  A             ST      ACØ,TIMER       ;STORE ACØ IN TIMER
ØØØ4  51ØØ  A             LI      AC1,Ø           ;CLEAR RPM COUNTER
ØØØ5  328Ø  A             SFLG    IE2             ;ENABLE IE2
ØØØ6  398Ø  A             SFLG    IEN             ;ENABLE INTERRUPTS
ØØØ7  Ø4ØØ  A   T1:       CFR     ACØ             ;COPY FLAGS TO ACØ
ØØØ8  46FE  A             BOC     6,T1            ;TIMER FINISHED?
ØØØ9  5C4Ø  A             RCPY    AC1,ACØ         ;YES, COPY AC1 TO ACØ
ØØØA  9C42  A             SKG     ACØ,MAX         ;SKIP IF COUNT > MAX
ØØØB  19Ø3  A             JMP     T2              ;
ØØØC  64ØØ  A             PULL    ACØ             ;RESTORE REGISTERS
ØØØD  65ØØ  A             PULL    AC1             ;
ØØØE  8ØØØ  A             RTS                     ;COUNT OVER RETURN
ØØØF  9Ø41  A   T2:       SUBB    ACØ,MIN         ;SUBTRACT MIN FROM COUNT
ØØ1Ø  42Ø3  A             BOC     2,T3            ;BRANCH IF COUNT OK
ØØ11  64ØØ  A             PULL    ACØ             ;RESTORE REGISTERS
ØØ12  65ØØ  A             PULL    AC1             ;
ØØ13  8ØØ1  A             RTS     1               ;COUNT UNDER RETURN
ØØ14  64ØØ  A   T3:       PULL    ACØ             ;RESTORE REGISTERS
ØØ15  65ØØ  A             PULL    AC1             ;
ØØ16  8ØØ2  A             RTS     2               ;COUNT OK RETURN
                ;         INTERRUPT 2 SERVICE ROUTINE
ØØ17  32ØØ  A   INTR2:    PFLG    IE2             ;RESET IE2
ØØ18  328Ø  A             SFLG    IE2             ;ENABLE IE2
ØØ19  338Ø  A             SFLG    IE3             ;ENABLE IE3
ØØ1A  ADØ8  A             DSZ     TIMER           ;DECREMENT TIMER
ØØ1B  7CØØ  A             RTI                     ;TIMER NOT ZERO
ØØ1C  32ØØ  A             PFLG    IE2             ;TIMER = Ø, RESET IE2
ØØ1D  33ØØ  A             PFLG    IE3             ;RESET IE3
ØØ1E  7CØØ  A             RTI                     ;RETURN
                ;         INTERRUPT 3 SERVICE ROUTINE
ØØ1F  33ØØ  A   INTR3:    PFLG    IE3             ;RESET IE3
ØØ2Ø  338Ø  A             SFLG    IE3             ;ENABLE IE3
ØØ21  79Ø1  A             AISZ    AC1,1           ;ADD 1 TO PULSE COUNT
ØØ22  7CØØ  A             RTI                     ;RETURN
ØØ23  ØØØØ  A   TIMER:    .WORD   Ø               ;TIME COUNTER
      ØØØØ                .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows a 16-bit binary counter interconnected with a 16-bit magnitude comparator to form a Modulo-N Divider (Operation of the binary counter and the magnitude comparator is covered on pp. 4-34 to 4-36 and 4-28 to 4-30.) For this application, counting is enabled when the externally-generated ENABLE signal is set low to allow the counter to continuously count up from zero to the value of the Dividy-By-N input to the comparator. When the output of the counter equals the Dividy-by-N value, the A = B output of the comparator goes high for approximately one clock pulse, and the counter is reset to zero on the positive-going edge of the next clock pulse to initiate another count cycle. The A > B output of the comparator ensures that circuit operation will not be affected should the counter output be preset to an illegal value when power is first turned on.



ASSIGNMENTS

The Modulo-N Divider function may be implemented with PACE as a double-entry subroutine. The flowchart and program listing that follow assume that one memory location is dedicated to storage of the count, a second memory location is dedicated to storage of the Divide-by-N value, that accumulator AC0 is used as a working register for altering the stored count, and that input/output assignments are as listed below.

INPUTS:

MODULO-N DIVIDER	PACE
A > B reset	RESET entry to Modulo-N Divider subroutine
Count (Enable, Clock)	MODULO entry to Modulo-N Divider subroutine (clock rate is equal to frequency of calling)
Divide-by-N value	Contents of memory-location PRESET

OUTPUTS:

MODULO-N DIVIDER	PACE
A = B	Status Register bit 7 (carry flag)
Counter output	Contents of memory-location COUNT

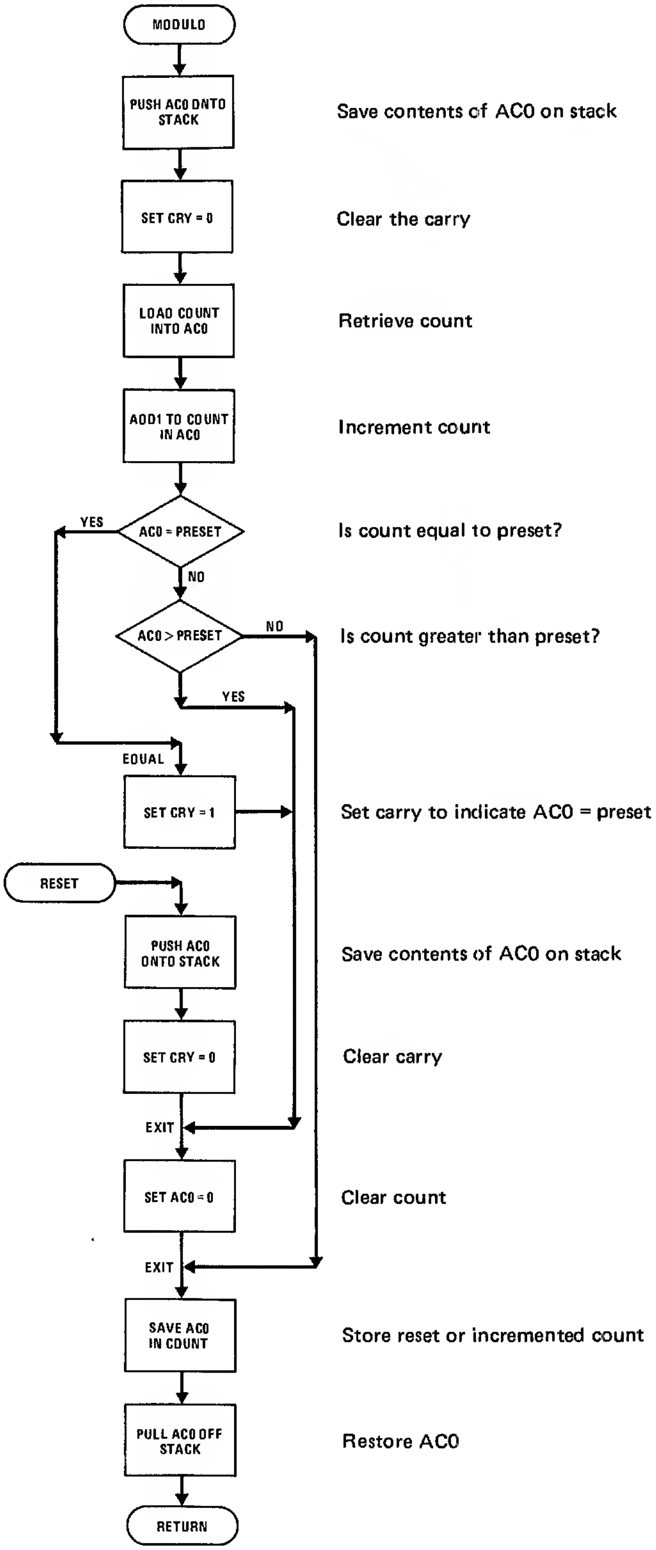
FUNCTIONAL OPERATION

This program is a double-entry subroutine that either resets or increments the Modulo-N counter (contents of memory-location COUNT). Since both entries to the subroutine employ AC0 as a working register, the original contents of AC0 are automatically saved on the stack at the start of the subroutine and restored at the end of the subroutine. For the RESET call, the carry flag is reset to clear any previous status after AC0 is saved on the stack (see the preface); AC0 is then set to zero and loaded into COUNT, which provides a starting value of zero for the first counting sequence. (Subsequent resetting to zero of the stored count occurs automatically at the completion of each counting sequence.)

For the MODULO call, the carry flag is reset, AC0 is loaded from COUNT after being saved on the stack, then AC0 is incremented by one and compared with the Divide-by-N value stored in memory-location PRESET. If the two values are equal, the carry flag is set high to indicate completion of a counting sequence; the contents of AC0 are then set to zero and loaded into COUNT, which provides a starting value of zero for the next counting sequence. If the two values are not equal, the contents of AC0 and PRESET are compared a second time to determine whether a greater value is preset in AC0. (This second test provides the same function as the A > B output of the comparator in the hardware configuration.)

If the value in AC0 is less than the value in PRESET, the carry remains low, and the contents of AC0 are returned to COUNT; this increments the stored count by one. A value in AC0 greater than the value in PRESET indicates an erroneous counting sequence. For this condition, the carry again remains low, and the contents of AC0 are set to zero and loaded into COUNT to initialize the stored count to zero. A new count sequence is initiated starting with the next subroutine call.

FLOW CHART



PROGRAM LISTING

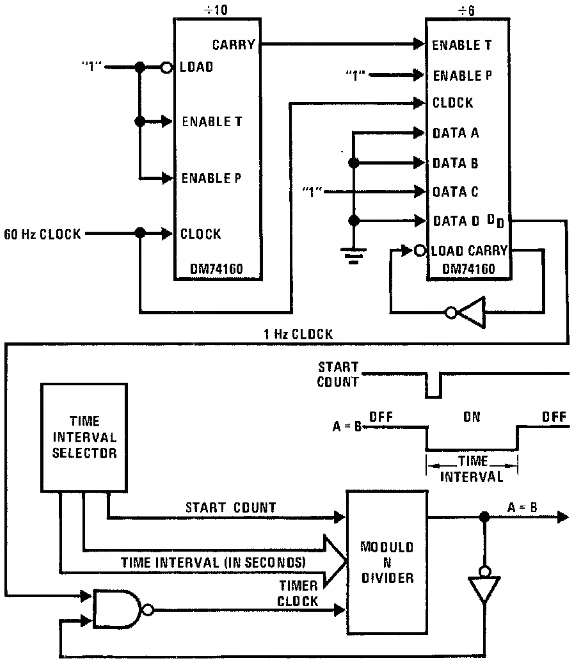
```
                      ;         MODULO N DIVIDER
        ØØØØ     ACØ        =         Ø
        ØØØ7     CRY        =         7
  ØØØØ  6ØØØ  A  MODULO:    PUSH      ACØ                ;SAVE ACØ ON STACK
  ØØØ1  37ØØ  A             PFLG      CRY                ;SET CARRY = Ø
  ØØØ2  C1ØE  A             LD        ACØ,COUNT          ;LOAD COUNT INTO ACØ
  ØØØ3  78Ø1  A             AISZ      ACØ,1              ;ADD 1 TO COUNT IN ACØ
  ØØØ4  F1ØD  A             SKNE      ACØ,PRESET         ;ACØ = PRESET VALUE?
  ØØØ5  19Ø3  A             JMP       EQUAL              ;YES
  ØØØ6  9DØB  A             SKG       ACØ,PRESET         ;ACØ > PRESET VALUE?
  ØØØ7  19Ø6  A             JMP       EXIT               ;NO
  ØØØ8  19Ø4  A             JMP       RESET+2            ;YES
  ØØØ9  378Ø  A  EQUAL:     SFLG      CRY                ;SET CARRY = 1
  ØØØA  19Ø2  A             JMP       RESET+2            ;
  ØØØB  6ØØØ  A  RESET:     PUSH      ACØ                ;SAVE ACØ ON STACK
  ØØØC  37ØØ  A             PFLG      CRY                ;SET CARRY = Ø
  ØØØD  5ØØØ  A             LI        ACØ,Ø              ;CLEAR ACØ
  ØØØE  D1Ø2  A  EXIT:      ST        ACØ,COUNT          ;SAVE ACØ IN COUNT
  ØØØF  64ØØ  A             PULL      ACØ                ;RESTORE ACØ FROM STACK
  ØØ1Ø  8ØØØ  A             RTS                          ;RETURN
  ØØ11  ØØØØ  A  COUNT:     .WORD     Ø                  ;CURRENT COUNT VALUE
  ØØ12  ØØØØ  A  PRESET:    .WORD     Ø                  ;PRESET VALUE
        ØØØØ                .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The diagram below shows a Modulo-N Divider interconnected with a time interval selector and a 1 Hz real-time clock to form an interval timer. This timer provides intervals that range from 1 second to approximately 18 hours (in 1 second increments).

For this application, the A = B output of the Modulo-N Divider goes low ("ON") when the counter is reset by the Start Count pulse, and it remains low until the counter is clocked up to the value of the Time Interval input. When the counter output equals the Time Interval input, the A = B output of the Modulo-N Divider goes high ("OFF") to disable the 1 Hz clock input to the counter, holding the interval timer in the "OFF" state until the next Start Count pulse is received.



ASSIGNMENTS

An interval timer function may be implemented with PACE as a subroutine that is called by the main program to select a real-time output ranging from one second to approximately 18.2 hours (in one second increments). The flowchart and program listing that follow assume that accumulator AC3 is used as an input data register and as a working register (for entering the desired time interval into a dedicated memory location, and keeping track of elapsed time, respectively), that the PACE Level 2 Interrupt input is continuously driven by a low-going 10μs clock pulse at a 60 Hz rate, and that input/output assignments are as listed below.

NOTE: The Level 2 Interrupt clock can be easily derived by buffering and squaring the 60 Hz line input, and edge detecting either the positive or negative alternation. For a detailed description of PACE interrupt signal requirements, refer to the material that starts on page 3-2.

INPUTS:

REAL-TIME CLOCK GENERATOR AND INTERVAL TIMER	PACE
60 Hz Clock	Level 2 Interrupt input continuously driven by low-going 10μs clock pulse at 60 Hz rate
Start Count	TIME ON entry to Interval Timer subroutine
Time Interval	Contents of AC3 when subroutine is called by main program via TIME ON entry

OUTPUTS:

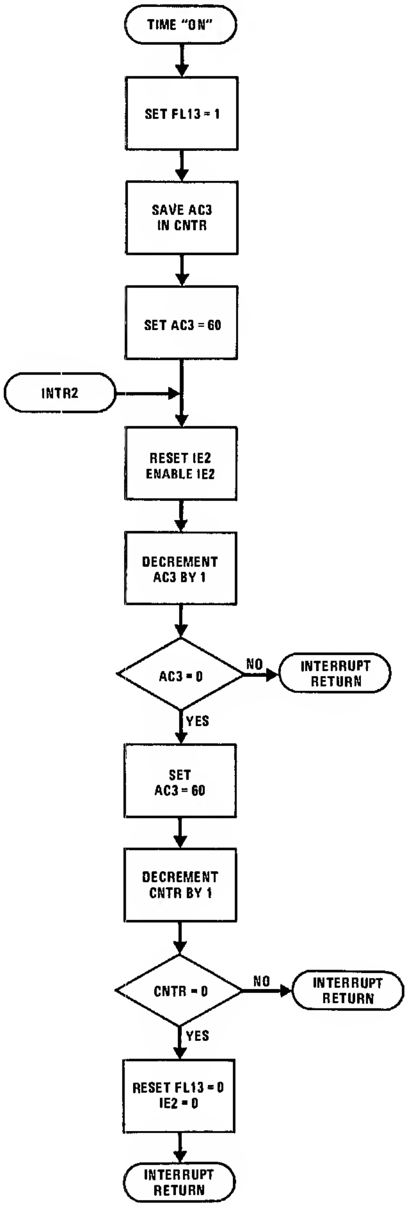
REAL-TIME CLOCK GENERATOR AND INTERVAL TIMER	PACE
A = B	Status Register bit 13 (flag 13)

FUNCTIONAL OPERATION

This program is a single-entry subroutine that causes the flag 13 output of PACE to be held set for a specific amount of time, which ranges from one second to approximately 18 hours (in one second intervals). It is assumed that when the subroutine is called by the main program the desired time interval has already been entered into AC3. After flag 13 is set upon entry to the subroutine the contents of AC3 are loaded into memory-location CNTR to control the amount of time that flag 13 remains set. AC3 is then set to 60, and the Level 2 Interrupts are enabled to allow the 60 Hz interrupt clock to be counted-down to the desired timer output.

Counting-down of the 60 Hz interrupt clock is accomplished by decrementing AC3 each time an interrupt is detected, until the contents of AC3 equal zero. Each time the contents of AC3 equal zero, AC3 is reset to 60 and the contents of CNTR are decremented by one. Thus, CNTR is decremented at a 1 Hz rate until it equals zero. When CNTR equals zero, flag 13 is reset to terminate the timer output, and Level 2 Interrupts are disabled to inhibit processing of the Level 2 Interrupt clock until the TIME "ON" subroutine is called again by the main program.

FLOW CHART



Set flag 13

AC3 contains time in seconds

Interrupt 2 entry point

Reset and enable interrupt 2

Test for zero value

Test value of CNTR (in main memory) for zero value. If not zero, return to main program; if zero, reset flag 13 to end timer cycle

Clear flag 13 and interrupt 2

Return to main program

PROGRAM LISTING

```
                     ;         REAL  TIME  CLOCK
       0002          IE2       =        2
       0003          AC3       =        3
       000D          FL13      =        13
  0000 3D80  A       TIMEON:   SFLG     FL13             ;SET FLAG = 1
  0001 DD0B  A                 ST       AC3,CNTR         ;SAVE SECONDS IN CNTR
  0002 533C  A                 LI       AC3,60           ;INITIALIZE AC3
  0003 3200  A       INTR2:    PFLG     IE2              ;RESET INTERRUPT 2
  0004 3280  A                 SFLG     IE2              ;ENABLE INTERRUPT 2
  0005 7BFF  A                 AISZ     AC3,-1           ;DECREMENT AC3 BY 1
  0006 7C00  A                 RTI                       ;NOT ZERO, RETURN
  0007 533C  A                 LI       AC3,60           ;SET AC3 TO 60 (DECIMAL)
  0008 AD04  A                 DSZ      CNTR             ;DECREMENT CNTR BY 1
  0009 7C00  A                 RTI                       ;NOT ZERO, RETURN
  000A 3D00  A                 PFLG     FL13             ;CNTR=0, RESET FL13=0
  000B 3200  A                 PFLG     IE2              ;RESET INTERRUPT 2
  000C 7C00  A                 RTI                       ;RETURN
  000D 0000  A       CNTR:     .WORD    0                ;COUNTER
       0000                    .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

PACE is readily adapted to pseudo-random number generation by the application of an asynchronous clock to one of the Branch Condition inputs. The diagram below shows a DM74C14 Schmitt Trigger configured as a square-wave generator, which drives the JC15 input of PACE. For this application, the Schmitt Trigger RC network is adjusted to cause the 16-bit numbers generated by PACE to appear random.

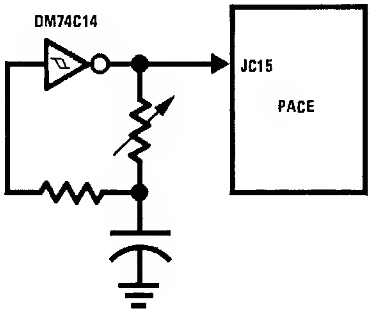
ASSIGNMENTS

The flowchart and program listing that follow assume that AC1 and AC0 are used as working and output data registers, respectively, and that an external oscillator is connected to the JC15 input of PACE.

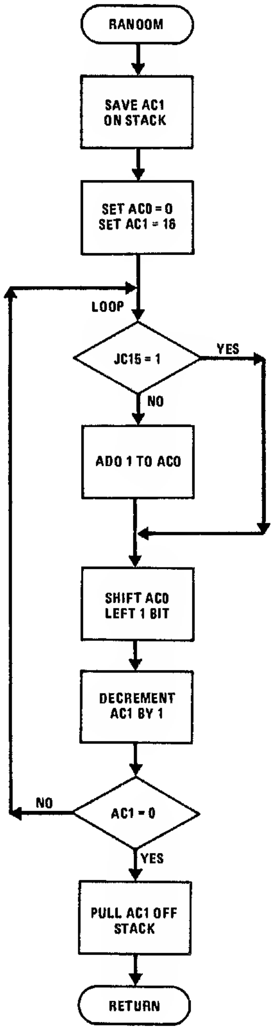
FUNCTIONAL OPERATION

This program is written as a single-entry subroutine that generates pseudo-random 16-bit numbers. The subroutine uses the PACE instruction execution time as a fixed clock, and processes the external oscillator input as a variable clock; the resulting 16-bit number generated is a function of the phase angle that exists between the two clocks at the start of the subroutine.

Since the subroutine requires that AC1 be used as a working register, the first operation of the subroutine is to store AC1 on the stack so that its original contents can be restored at the end of the subroutine. After AC1 is stored on the stack, AC0 is initialized to 0 for use as a random-number generator and AC1 is initialized to 16 for use as a loop counter. The JC15 input to PACE is then tested via a Branch-On-Condition (BOC) instruction to cause AC0 to be incremented and/or shifted left one bit at a time. After each shift, the contents of AC1 are decremented by one and tested for zero. When the contents of AC1 equal zero, AC0 contains a randomly-generated 16-bit number. AC1 is then pulled from the stack to restore its original contents and the subroutine is exited with the randomly-generated 16-bit number stored in AC0.



FLOW CHART



Save contents of AC1 on stack

AC1 is the loop count

Flag high?

Finished?

Restore AC1

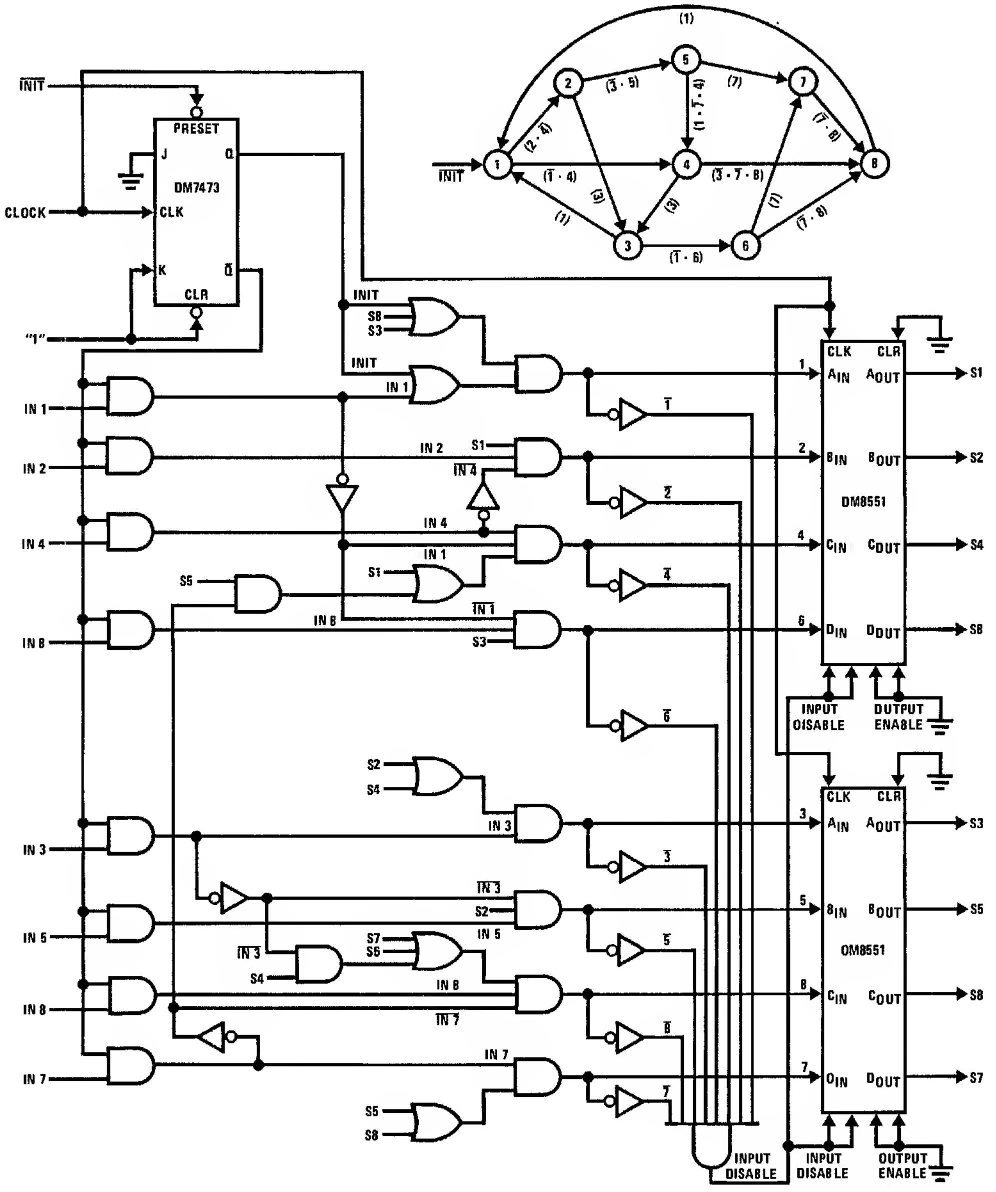
PROGRAM LISTING

```
                 ;        PSEUDO-RANDOM NUMBER GENERATOR
       ØØØØ      ACØ      =       Ø
       ØØØ1      AC1      =       1
ØØØØ   61ØØ  A   RANDOM:  PUSH    AC1              ;SAVE AC1 ON STACK
ØØØ1   5ØØØ  A            LI      ACØ,Ø            ;CLEAR ACØ
ØØØ2   511Ø  A            LI      AC1,16           ;SET LOOP COUNTER
ØØØ3   4FØ1  A   LOOP:    BOC     15,R1            ;BRANCH IF JC15 = 1
ØØØ4   78Ø1  A            AISZ    ACØ,1            ;ADD 1 TO ACØ
ØØØ5   28Ø2  A   R1:      SHL     ACØ,1,Ø          ;SHIFT ACO LEFT 1 BIT
ØØØ6   79FF  A            AISZ    AC1,-1           ;DECREMENT AC1 BY 1
ØØØ7   19FB  A            JMP     LOOP             ;
ØØØ8   65ØØ  A            PULL    AC1              ;RESTORE AC1 FROM STACK
ØØØ9   8ØØØ  A            RTS                      ;RETURN
       ØØØØ               .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

The logic and state diagrams below show how an 8-bit binary input may be decoded for state sequencing. Operation of the logic is controlled by the Initialize input. When the Initialize input goes low, the Enable flip-flop is preset to force the State-1 output of the decode logic high, and the States 2 through 8 outputs low; this allows the State Register to be initialized to State 1 on the first positive alternation of the clock. When the Initialize input is returned high, the Enable flip-flop is clocked reset on the next positive alternation of the clock to enable normal operation of the decode logic. While enabled, the decode logic continually compares the 8-bit binary input with the output of the State Register to detect a valid state change as specified by the sequence chart. For example, following initialization the high S1 output of the State Register enables the decode logic to provide a high State-2 output when input bit 2 is high and input bit 4 is low, or a high State 4 output when input bit 1 is low and input bit 4 is high; any other combination of inputs results in all eight outputs of the decode logic being low. Sampling of the decode logic output occurs on the positive-going edge of each clock pulse. If one of the eight possible outputs of the decode logic is high, the Input Disable signal will be low and the State Register will be clocked to the new state. If all eight outputs of the decode logic are low, the Input Disable signal will be high and the State Register will be inhibited from changing state.



ASSIGNMENTS

The state sequencer function may be implemented with PACE as a single-entry subroutine. The flowchart and program listing that follow assume that a memory location is dedicated to storage of the current state, that accumulator AC2 is used as a working register for detecting the current state, accumulator AC0 is used as an input data register and as a working register (for entering the 8-bit state-sequence word and changing the stored state accordingly), and that input/output assignments are as listed below.

INPUTS:

STATE SEQUENCER LOGIC	PACE
Initialize	Main program storage of State 1 (X'0001) in memory-location STATE
Data	AC0 Bit
I1	0
I2	1
.	.
.	.
.	.
I8	7
State Clock	STATE entry to State Sequencer subroutine

OUTPUTS:

STATE SEQUENCER LOGIC	PACE
S1	Contents of memory-location STATE = X'0001
S2	Contents of memory-location STATE = X'0002
S3	Contents of memory-location STATE = X'0003
S4	Contents of memory-location STATE = X'0004
S5	Contents of memory-location STATE = X'0005
S6	Contents of memory-location STATE = X'0006
S7	Contents of memory-location STATE = X'0007
S8	Contents of memory-location STATE = X'0008

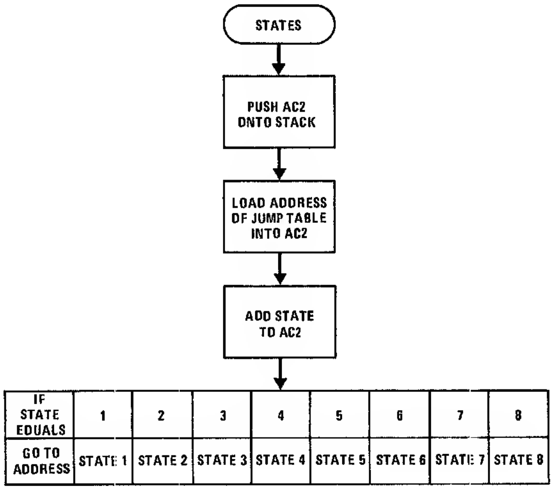
FUNCTIONAL OPERATION

This program is written as a single-entry subroutine that processes an 8-bit state sequence input. When the subroutine is called, it is assumed that the state sequence input has already been loaded into AC0. The first step of the subroutine, therefore, is to push working register AC2 onto the stack so that the original contents of AC2 can be restored at the end of the subroutine. After AC2 is pushed onto the stack, the address of the State Jump table (JMPTBL) is loaded into AC2; AC2 is then incremented by the value stored in memory-location STATE to cause the subroutine to branch to the corresponding STATE routine. For example, if the value in memory-location STATE is X'0001, the subroutine will branch to the STATE 1 routine; if the value is X'0002 the subroutine will branch to the STATE 2 routine; and so forth.

The States 1-8 routines are functionally identical in that each routine sequentially tests appropriate bits of the State Sequence input (stored in AC0) to determine whether a valid state change is indicated. Testing of the state sequence input bits is accomplished by rotating AC0 right or left as required to locate each significant bit at AC0 bit position 0, 1, 2, or 15, then employing Branch-On-Condition (BOC) instructions to detect the logic states of the significant bits. If a valid state change is indicated, a branch to an appropriate SET routine loads the new state into AC0; if a valid state change is not indicated, the branch path is to the EXIT and the current state is loaded into AC0 from STATE. After the new or current state is loaded into AC0, AC2 is pulled from the stack to restore the original contents, and AC0 is stored in STATE to update or retain the stored output.

Upon return to the main program, the State Output will be present both in AC0 and STATE. Thus, the main program can detect the output state by decrementing AC0 and using Branch-On-Condition (BOC) instructions to select an appropriate branch path for the main program when the contents of AC0 equal zero.

FLOW CHART



IF STATE EQUALS	1	2	3	4	5	6	7	8
GO TO ADDRESS	STATE 1	STATE 2	STATE 3	STATE 4	STATE 5	STATE 6	STATE 7	STATE 8

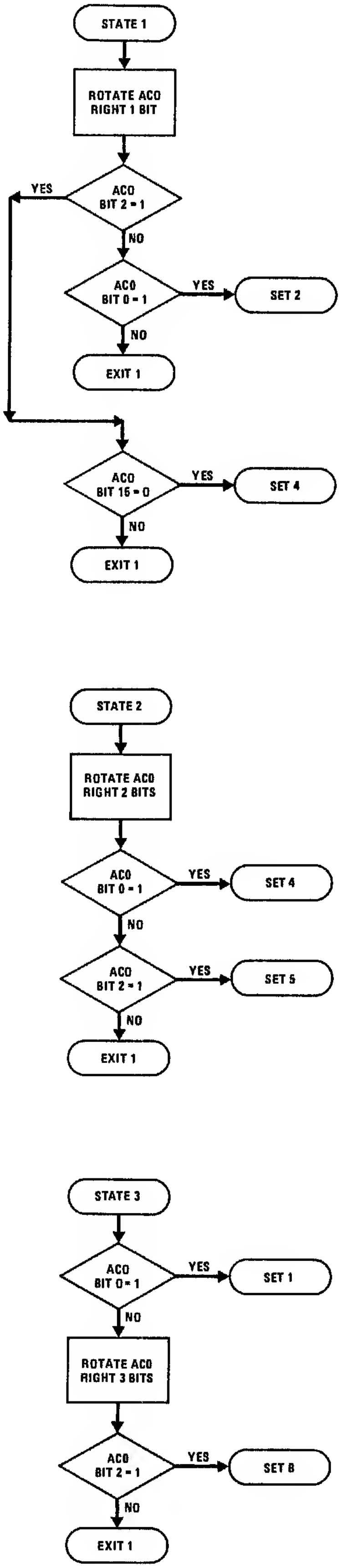
Save contents of AC2 on stack

Load the beginning address of the jump table into AC2

Add the state for a displacement into the jump table

Jump to appropriate state routine

FLOW CHART (Continued)



Position input 2 at bit 0

Input 4 = 1?

Input 2 = 1?

Input 1 = 0?

Position input 3 at bit 0

Input 3 = 1?

Input 5 = 1?

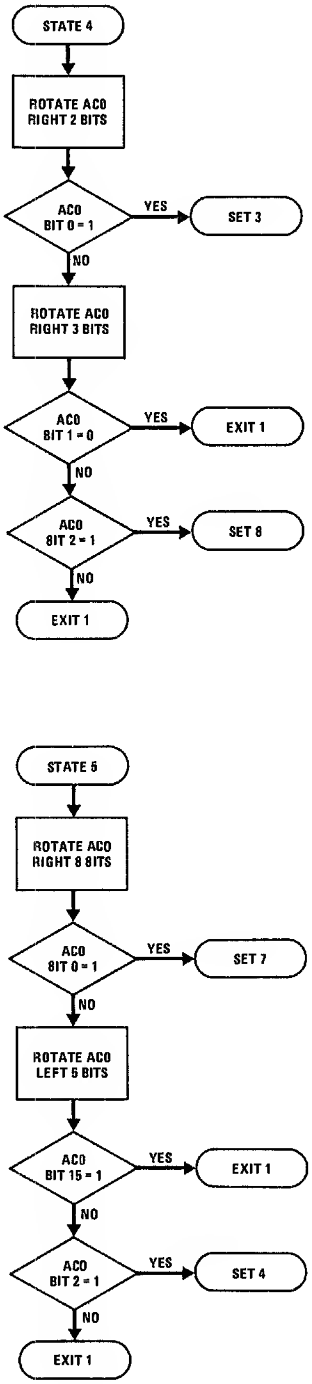
Don't change state

Input 1 = 1?

Position input 6 at bit 2

Input 6 = 1?

FLOW CHART (Continued)



Position input 3 at bit 0

Input 3 = 1?

Position input 7 at bit 1

Input 7 = 1?

Input 8 = 1?

Position input 7 at bit 0

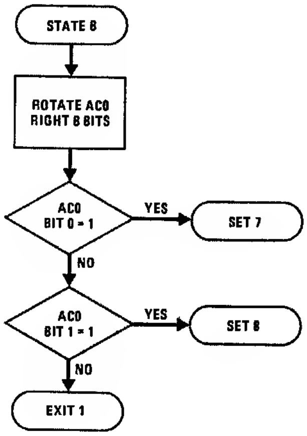
Input 7 = 1?

Position input 1 at bit 15

Input 1 = 1?

Input 4 = 1?

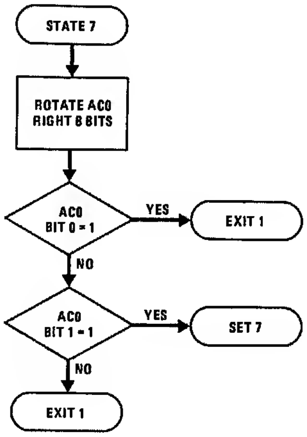
FLOW CHART (Continued)



Position input 7 at bit 0

Input 7 = 1?

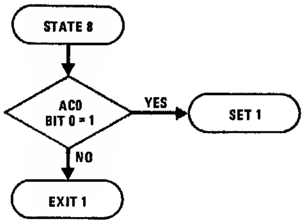
Input 8 = 1?



Position input 7 at bit 0

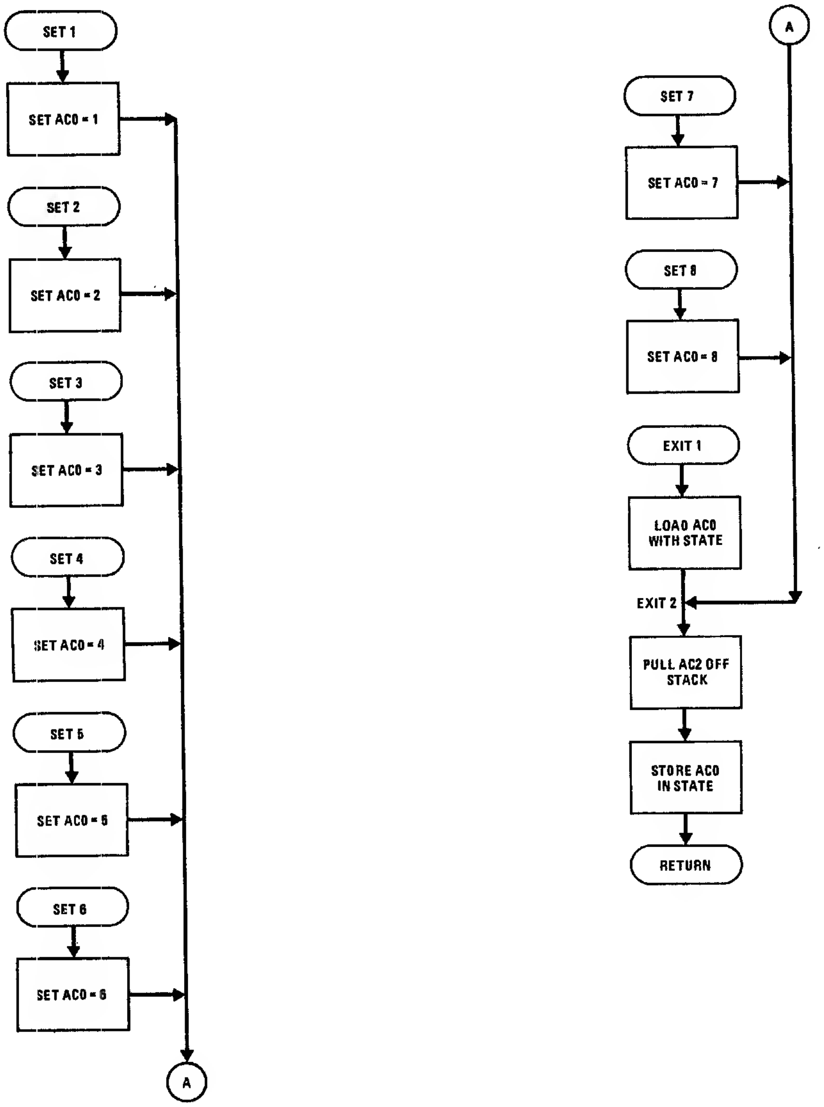
Input 7 = 1?

Input 8 = 1?



Input 1 = 1?

FLOW CHART (Continued)



Restore AC2

PROGRAM LISTING

```
                  ;        STATE TEST SEQUENCER
     0000         AC0      =      0
     0002         AC2      =      2
0000 6200 A       STATES:  PUSH   AC2             ;SAVE AC2 ON STACK
0001 C902 A                LD     AC2,JMPTBL      ;LOAD JUMP TABLE ADDRESS
0002 E942 A                ADD    AC2,STATE       ;ADD STATE
0003 1A00 A                JMP    0(AC2)          ;JUMP TO A STATE ROUTINE
0004 0004 T       JMPTBL:  .WORD  .               ;JUMP TABLE
0005 1907 A                JMP    STATE1
0006 190C A                JMP    STATE2
0007 190F A                JMP    STATE3
0008 1912 A                JMP    STATE4
0009 1917 A                JMP    STATE5
000A 191C A                JMP    STATE6
000B 191F A                JMP    STATE7
000C 1922 A                JMP    STATE8
                  ;
000D 2402 A       STATE1:  ROR    AC0,1,0         ;MOVE INPUT 2 TO BIT 0
000E 4602 A                BOC    6,ST1A          ;BRANCH IF INPUT 4 = 1
000F 4323 A                BOC    3,SET2          ;BRANCH IF INPUT 2 = 1
0010 1930 A                JMP    EXIT1           ;DON'T CHANGE STATE
0011 4225 A       ST1A:    BOC    2,SET4          ;BRANCH IF INPUT 1 = 0
0012 192E A                JMP    EXIT1           ;DON'T CHANGE STATE
0013 2404 A       STATE2:  ROR    AC0,2,0         ;MOVE INPUT 3 TO BIT 0
0014 4322 A                BOC    3,SET4          ;BRANCH IF INPUT 3 = 1
0015 4623 A                BOC    6,SET5          ;BRANCH IF INPUT 5 = 1
0016 192A A                JMP    EXIT1           ;DON'T CHANGE STATE
0017 4319 A       STATE3:  BOC    3,SET1          ;BRANCH IF INPUT = 1
0018 2406 A                ROR    AC0,3,0         ;MOVE INPUT 6 TO BIT 2
0019 4621 A                BOC    6,SET6          ;BRANCH IF INPUT 6 = 1
001A 1926 A                JMP    EXIT1           ;DON'T CHANGE STATE
001B 2404 A       STATE4:  ROR    AC0,2,0         ;MOVE INPUT 3 TO BIT 0
001C 4318 A                BOC    3,SET3          ;BRANCH IF INPUT 3 = 1
001D 2406 A                ROR    AC0,3,0         ;MOVE INPUT 7 TO BIT 1
001E 4422 A                BOC    4,EXIT1         ;EXIT IF INPUT 7 = 1
001F 461F A                BOC    6,SET8          ;BRANCH IF INPUT 8 = 1
0020 1920 A                JMP    EXIT1           ;DON'T CHANGE STATE
0021 240C A       STATE5:  ROR    AC0,6,0         ;MOVE INPUT 7 TO BIT 0
0022 431A A                BOC    3,SET7          ;BRANCH IF INPUT 7 = 1
0023 200A A                ROL    AC0,5,0         ;MOVE INPUT 1 TO BIT 15
0024 4B1C A                BOC    11,EXIT1        ;EXIT IF INPUT 1 = 1
0025 4611 A                BOC    6,SET4          ;BRANCH IF INPUT 4 = 1
0026 191A A                JMP    EXIT1           ;DON'T CHANGE STATE
0027 240C A       STATE6:  ROR    AC0,6,0         ;MOVE INPUT 7 TO BIT 0
0028 4314 A                BOC    3,SET7          ;BRANCH IF INPUT 7 = 1
0029 4415 A                BOC    4,SET8          ;BRANCH IF INPUT 8 = 1
002A 1916 A                JMP    EXIT1           ;DON'T CHANGE STATE
002B 240C A       STATE7:  ROR    AC0,6,0         ;MOVE INPUT 7 TP BIT 0
002C 4314 A                BOC    3,EXIT1         ;EXIT IF INPUT 7 = 1
002D 4411 A                BOC    4,SET8          ;BRANCH IF INPUT 8 = 1
002E 1912 A                JMP    EXIT1           ;DON'T CHANGE STATE
002F 4301 A       STATE8:  BOC    3,SET1          ;BRANCH IF INPUT 1 = 1
0030 1910 A                JMP    EXIT1           ;DON'T CHANGE STATE
                  ;
0031 5001 A       SET1:    LI     AC0,1           ;SET STATE = 1
0032 190F A                JMP    EXIT2           ;
0033 5002 A       SET2:    LI     AC0,2           ;SET STATE = 2
```

PROGRAM LISTING (Continued)

```
0034 190D A          JMP     EXIT2          ;
0035 5003 A  SET3:   LI      AC0,3          ;SET STATE = 3
0036 190B A          JMP     EXIT2          ;
0037 5004 A  SET4:   LI      AC0,4          ;SET STATE = 4
0038 1909 A          JMP     EXIT2          ;
0039 5005 A  SET5:   LI      AC0,5          ;SET STATE = 5
003A 1907 A          JMP     EXIT2          ;
003B 5006 A  SET6:   LI      AC0,6          ;SET STATE = 6
003C 1905 A          JMP     EXIT2          ;
003D 5007 A  SET7:   LI      AC0,7          ;SET STATE = 7
003E 1903 A          JMP     EXIT2          ;
003F 5008 A  SET8:   LI      AC0,8          ;SET STATE = 8
0040 1901 A          JMP     EXIT2          ;
0041 C103 A  EXIT1:  LD      AC0,STATE      ;LOAD STATE INTO AC0
0042 6600 A  EXIT2:  PULL    AC2            ;RESTORE AC2 FROM STACK
0043 D101 A          ST      AC0,STATE      ;STORE AC0 IN STATE
0044 8000 A          RTS                    ;RETURN
0045 0001 A  STATE:  .WORD   1
     0000            .END
```

HARDWARE SUMMARY AND PROGRAM DESCRIPTION

SUMMARY

A switch-bounce-detect function can be implemented with PACE using a combination of hardware (for entry of the switch data) and an interrupt service routine (for detection of switch bounce). The basic functions of the hardware configuration are the generation of a Level 5 Interruput output to PACE each time that a switch setting is changed, and the routing of the switch data to PACE when the TRI-STATE® switch buffers are addressed in the ensuing interrupt service routine.

Generation of the Level 5 Interrupt is accomplished by WIRE-ORing the outputs of three 6-bit DM8136 comparators together to form an EXCLUSIVE-OR gate; this gate continually compares the logic level present at each T input with the logic level present at each corresponding B input. Each time that a switch setting is changed, the resultant change in logic level will be felt immediately at the T input, but not at the B input until the RC network charges to the new value. Thus, each change in switch setting will cause the EXCLUSIVE-OR gate to generate a low-level Interrupt 5 pulse that is equal in duration to the charge time of the RC network. Since PACE timing requirements may vary with system application, the values for the RC networks are typically chosen to yield a Level 5 Interrupt pulse that is slightly greater than one clock period in duration.

NOTE: For a detailed description of PACE interrupt signal requirements, refer to the material that starts on page 3-2.

Upon detection of the Level 5 Interrupt pulse, PACE executes an interrupt service routine that reads-in the switch data twice (at N ms intervals), then compares the two inputs to determine whether a valid data input was received the first time. If the two inputs are the same, PACE stores the switch data in a memory location for entry into the main program, then pulses the Flag 14 output to provide a "data accepted" indication via the one-shot timer and display circuits. If the two inputs are different, memory storage of the switch data and the "data accepted" indication are inhibited.

Execution of Load (LD)-from-address-X'8XXX (address bit 15 high) instructions, which clock the Bus Enable flip-flop set at NADS (address strobe) time, reads-in the switch data. While the Bus Enable flip-flop is set, the Q and $\overline{Q}$ outputs enable the TRI-STATE switch buffers and disable the memory and peripheral data buffers; this applies the switch data to PACE over the data bus. The instructions that follow the Load-from-address-X'8XXX instructions then reference memory or peripheral addresses below X'8XXX (address bit 15 low) to clock the Bus Enable flip-flop reset, and thereby reinstate normal communications between PACE, memory, and peripherals. Similarly, the NINIT input to the Bus Enable flip-flop ensures that the flip-flop will be reset when power is first applied, to allow execution of the power-up routine stored in memory.

ASSIGNMENTS

The flowchart and program listing provided for the Switch Bounce Detect, Level 5 Interrupt service routine assume that a memory location is dedicated to storage of valid switch data, that AC0 and AC1 are employed as input-data and working registers for entry and comparison of the initial and time-buffered switch data inputs, and that a pulsed Flag 14 output is provided to the one-shot time and display circuit for each valid switch-data entry.

FUNCTIONAL OPERATION

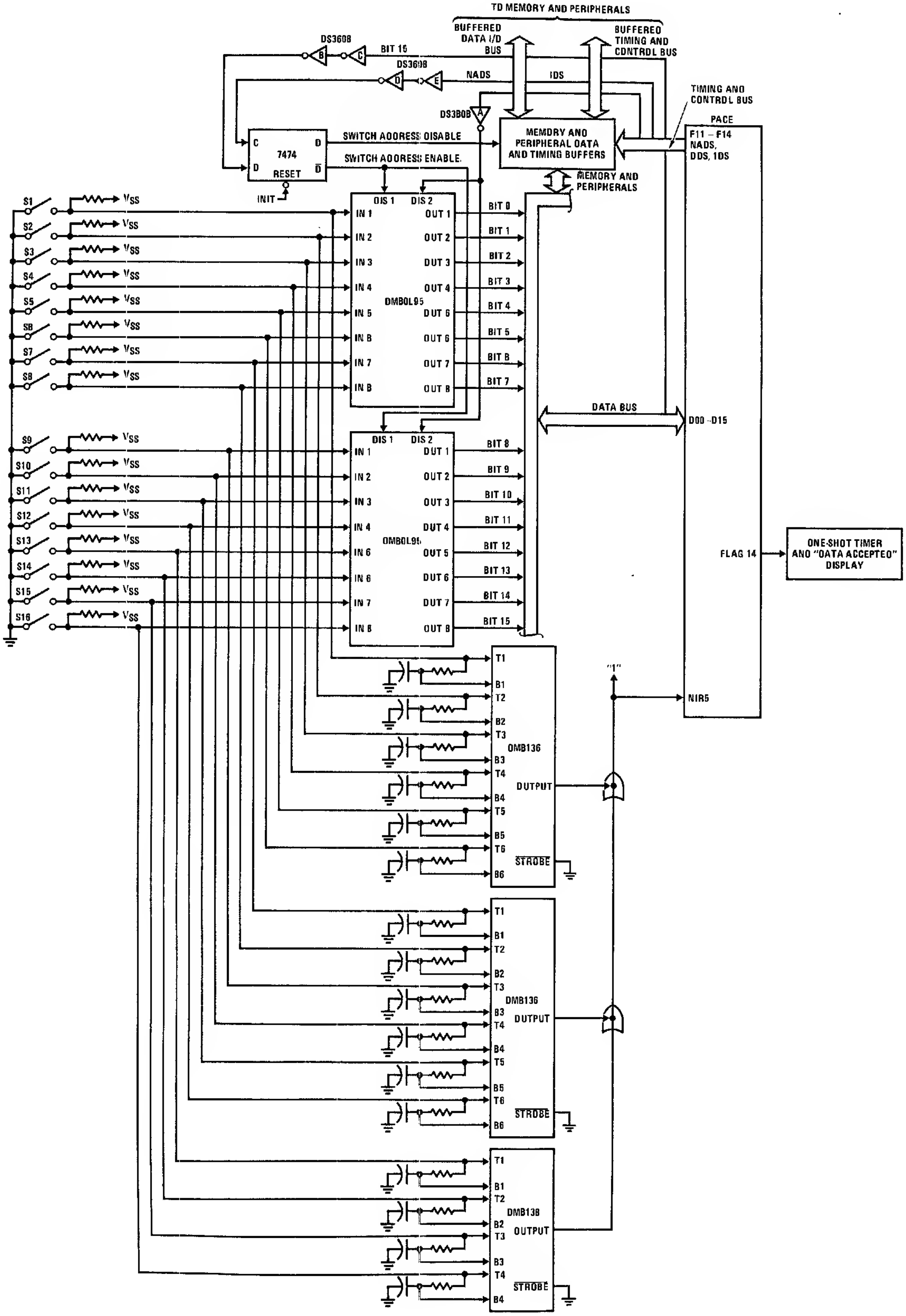
This program is written as a Level 5 Interrupt service routine; it is executed each time that a Level 5 Interrupt is detected following a change in switch setting. Since the service routine requires the use of AC0 and AC1 both as input-data and working registers, the first step of the routine is to save AC0 and AC1 on the stack so that the original contents can be restored at the end of the routine. After AC0 and AC1 are saved on the stack, a load (LD) instruction is executed for initial entry of the switch data into AC0. The switch data is then copied into AC1, and the preselected delay interval stored in memory-location MSECS is loaded in memory-location CNTR via AC0. Following this, the contents of AC0 are set to 51_{10} and decremented by one at a 19μs rate to provide a 1 ms delay cycle. When the contents of AC0 equal zero, the delay value stored in CNTR is decremented by one and the "delay cycle/decrement CNTR sequence" is repeated until the contents of CNTR equal zero.

Decrementing of AC0 at a 19μs rate is accomplished via an AISZ−1 instruction followed by a JMP −1 instruction. While AC0 is being decremented to zero, execution times for the AISZ and JMP instructions are 10.5μs and 8.5μs respectively. Upon detection of AC0 = 0, the AISZ instruction execution time increases to 12.5μs to provide an automatic skip to the instruction following the JMP −1 instruction. Thus, a DSZ instruction (15.5μs or 17.5μs for a CNTR > 0 or = 0, respectively) is executed to decrement the contents of CNTR by one. If the new value in CNTR is not zero, the JMP LOOP instruction (8.5μs execution time) following the DSZ instruction causes the service routine to loop back to the instruction, which sets AC1 = 51_{10} thereby enabling another delay cycle/decrement counter sequence.

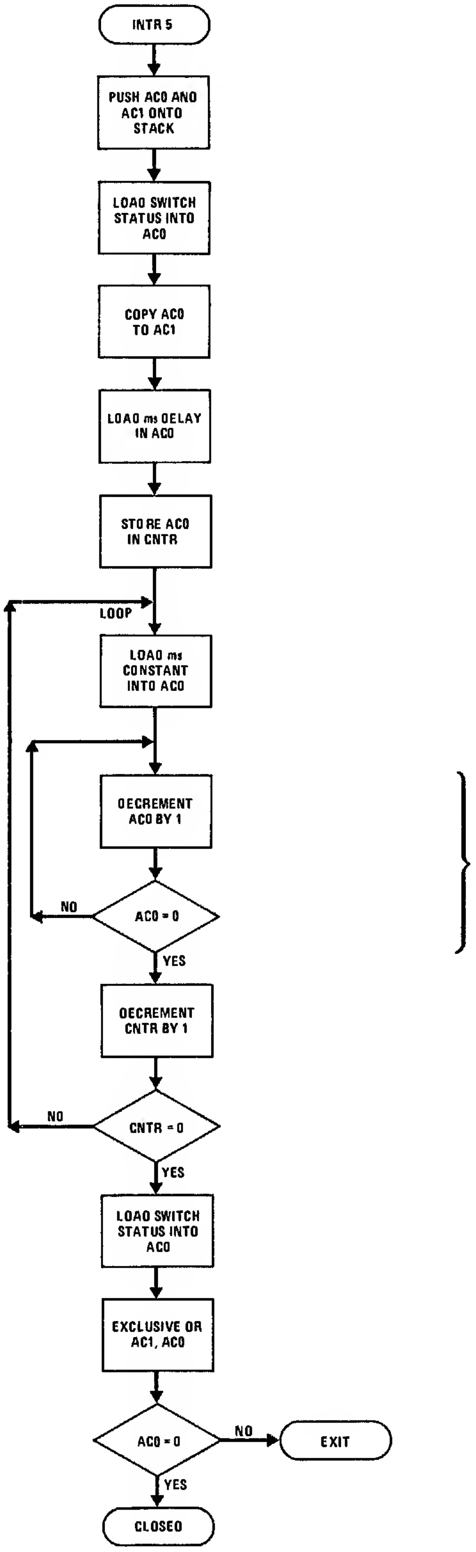
When the contents of CNTR are subsequently decremented to zero, the JMP LOOP instruction that follows the DSZ instruction is skipped, and a Load (LD) AC0 switch instruction is executed to enter the time-buffered switch data into AC0. The contents of AC1 (initial switch data entry) and AC0 are then EXCLUSIVE-OR'ed and the result is tested for zero via a Branch-On-Condition (BOC) instruction to determine whether the initial and time-buffered switch data inputs are the same.

If the two inputs are the same, the contents of AC0 will be zero, flag 14 is pulsed, the new switch data input is stored in memory-location STATUS. AC0 and AC1 will then be pulled from the stack to restore their original contents, Level 5 Interrupts will be reenabled by first resetting, then setting, the Level 5 Interrupt enable flag, and a Return From Interrupt (RTI) instruction will be executed to allow a return to the main program at the point where it was interrupted.

If the initial and time-buffered switch data entries are different, the contents of AC0 will not be zero, and the BOC instruction will reference the EXIT branch to skip over the Pulse Flag 14 and Save-AC1-in-Status instructions. Thus, the return to the main program will occur with the previous switch data entry stored in STATUS.



FLOW CHART



Interrupt-5 service routine

Contents of AC0 and AC1 on stack

First read of switch status

1 ms timing loop

Decrement ms count by 1

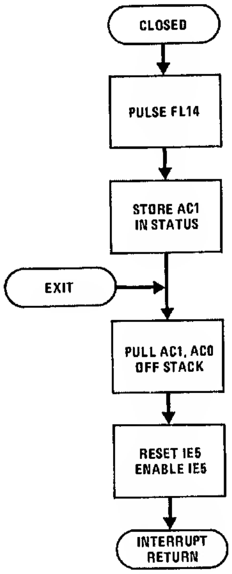
Is the count = 0?

Second read of switch status

Compare the first and second
readings of the switches

Were they the same?

FLOW CHART (Continued)



Indicate switch closure

Save the switch status

Restore AC1 and ACO

Reset and enable interrupt 5

PROGRAM LISTING

```
                      ;         SWITCH  DEBOUNCE
        ØØØØ          ACØ       =       Ø
        ØØØ1          AC1       =       1
        ØØØ5          IE5       =       5              ;INTERRUPT 5
        ØØØE          FL14      =       14             ;FLAG 14
                      ;         INTERRUPT 5 SERVICE ROUTINE
ØØØØ    6ØØØ  A       INTR5:    PUSH    ACØ            ;SAVE REGISTERS ON STACK
ØØØ1    61ØØ  A                 PUSH    AC1            ;
ØØØ2    A115  A                 LD      ACØ,@SWITCH    ;LOAD SWITCH STATUS
ØØØ3    5DØØ  A                 RCPY    ACØ,AC1        ;COPY ACØ TO AC1
ØØØ4    C111  A                 LD      ACØ,MSECS      ;LOAD NUMBER OF MILISECS
ØØØ5    D1ØF  A                 ST      ACØ,CNTR       ;STORE MILISECS IN CNTR
ØØØ6    5Ø33  A       LOOP:     LI      ACØ,51         ;LOAD MILISEC CONSTANT
ØØØ7    78FF  A                 AISZ    ACØ,-1         ;DECREMENT ACØ BY 1
ØØØ8    19FE  A                 JMP     .-1            ;AC1 NOT ZERO
ØØØ9    ADØB  A                 DSZ     CNTR           ;DECREMENT MILISEC COUNT
ØØØA    19FB  A                 JMP     LOOP           ;CNTR NOT ZERO
ØØØB    A1ØC  A                 LD      ACØ,@SWITCH    ;LOAD SWITCH STATUS
ØØØC    584Ø  A                 RXOR    AC1,ACØ        ;COMPARE NEW TO OLD
ØØØD    45Ø2  A                 BOC     5,EXIT         ;EXIT IF NEW NOT = OLD
ØØØE    3EØØ  A       CLOSED:   PFLG    FL14           ;INDICATE SWITCH CLOSURE
ØØØF    D5Ø7  A                 ST      AC1,STATUS     ;SAVE THE SWITCH STATUS
ØØ1Ø    65ØØ  A       EXIT:     PULL    AC1            ;RESTORE REGISTERS
ØØ11    64ØØ  A                 PULL    ACØ            ;
ØØ12    35ØØ  A                 PFLG    IE5            ;RESET INTERRUPT 5
ØØ13    358Ø  A                 SFLG    IE5            ;ENABLE INTERRUPT 5
ØØ14    7CØØ  A                 RTI                    ;RETURN
ØØ15    ØØØØ  A       CNTR:     .WORD   Ø              ;TIMER COUNTER
ØØ16    ØØØA  A       MSECS:    .WORD   1Ø             ;NUMBER OF MILISECS DELA
ØØ17    ØØØØ  A       STATUS:   .WORD   Ø              ;SWITCH STATUS SAVE
ØØ18    8ØØØ  A       SWITCH:   .WORD   Ø8ØØØ          ;ADDRESS OF SWITCHES
        ØØØØ                    .END
```

APPENDICES

APPENDIX A – GLOSSARY

ACCUMULATOR: Specifically, a data storage device (register) for work in progress; part of the equipment in the arithmetic unit of a processor, in which arithmetical and logical operations are performed (the **ALU**).

ADDRESS: A number that designates a register, a memory location, or a device.

ADDRESS FIELD: That part of an instruction or word containing an address or operand.

ASSEMBLER: A program that translates symbolic language to machine language.

BINARY: Involving a choice or condition of two alternatives (yes/no; on/off); a number system using the base 2.

BIT: Binary digit.

BUFFER: An area of memory that is used as a work area or to store data for an input/output operation.

BUS: A circuit over which data or power is transmitted.

BYTE: A group of consecutive binary digits usually operated upon as a unit.

CARRY: A condition occurring during addition when the sum of two digits equals or exceeds the number base; or, the digit to be added to the next higher column as a result of the sum overflow.

CENTRAL PROCESSING UNIT (CPU): The portion of any computer that consists of the arithmetic unit, the control unit, and the storage unit.

CLOCK: A master timing device used to provide the basic sequence pulses for the operation of a synchronous computer.

COMPILER: A program that produces a machine-language program from a source-language program.

COMPLEMENT: In the binary number system there are two complements: the "ones complement," and the "twos complement." The ones complement is obtained by converting all ones to zeros, and all zeros to ones. The twos complement may be obtained by first converting a binary number to its ones-complement and then adding one to the ones-complement. In binary logic, signals may be in one of two possible states: *true* or *false, high* or *low, on* or *off.* Thus, a signal is complemented by changing it from one state to the other state.

CONDITIONAL BRANCH: A branch that occurs only if a certain condition is present in the machine at the time the instruction is executed.

CONSOLE: The portion of the processor that may be used to control the machine manually, correct errors, determine the status of registers, counters, and storage, and manually revise the contents of storage.

CONTROL SECTION: The part of a processor that determines the interpretation and execution of instructions in their proper sequence, including the decoding of each instruction and the application of the proper signals to the registers, arithmetic and logic units in accordance with the decoded information.

DATA: A general term loosely used to denote any or all facts, numbers, letters, and symbols that can be processed or produced by a processor.

DEBUG: To isolate and remove malfunctions from a computer or mistakes from a program; also, a utilities program that helps correct application programs.

DIAGNOSTIC ROUTINE: A specific routine designed to locate either a malfunction in the processor or a mistake in coding.

EFFECTIVE ADDRESS: The addition of the contents of the base register and displacement plus, in some cases, the index register contents to form the address actually used in addressing main memory.

ENABLE: Restoration of a suppressed interrupt.

EXECUTE: To carry out an instruction or perform a routine.

FLAG: A bit used to indicate the status of an element.

FETCH: To retrieve a word of data from main memory.

FIRMWARE: Read-only memory (ROM), or the data or instructions stored in ROM.

HALT: A machine instruction that stops the execution of a program.

HEXADECIMAL: Related to a number system that uses the base 16.

HARDWARE: The physical equipment of the processor.

INDEX REGISTER: A register that modifies the operand address in an instruction or base address to yield a new effective address.

INITIALIZE: A program or hardware circuit that clears registers and sets counters and switches to their starting values.

INSTRUCTION: A user-coded macroinstruction that causes the microinstructions to perform certain operations.

INTERRUPT: A break in the normal flow of a system such that the flow can be resumed from that point at a later time. An interrupt is usually caused by a signal from an external source.

JUMP: An instruction or signal that, conditionally or unconditionally, specifies the location of the next instruction and directs the processor to that instruction.

LABEL: An ordered set of characters used to symbolically identify an instruction, an address, or à value.

LIST: An ordered set of items.

MACHINE LANGUAGE: The system of (binary) codes by which instructions and data are represented internally within a data processing system.

MACROINSTRUCTION: In general, any single instruction that causes a complete sequence of events to occur; a single instruction made up of a number of microinstructions that together perform a specific operation. A microinstruction is carried out in one microcycle.

MAIN MEMORY: Read/write memory that is external to the control ROM but is internal to the microprocessor.

MICROCYCLE: The basic machine cycle of the microprocessor.

MICROCODE: The steps or microinstructions of a microprogram, or the binary coded data contained in the microinstruction words of the control ROM.

MICROINSTRUCTION: See MACROINSTRUCTION.

MICROPROGRAM: A set of basic instructions (microinstructions) stored in read-only memory, programmable read-only memory, or read/write memory, and used by the control section of a processor to command registers, arithmetic and logic units.

MICROPROGRAMMING: Machine-language coding in which the coder builds his own machine instruction from the primitive basic instructions built into the hardware.

MNEMONICS: Operation codes written in easily-remembered symbolic code rather than the actual machine code.

OPERANDS: Any quantities entering or arising in an operation. An operand may be an argument, a result, a parameter, or an indication of the location of the next instruction.

OVERFLOW: The condition that arises, in a digital computer, when the result of an arithmetic operation exceeds the capacity of the storage space allotted.

PROGRAM: A group of related routines that solve a given problem.

PROGRAM COUNTER: A counter constructed in hardware that contains the address of the next instruction to be executed.

READ-ONLY MEMORY (ROM): A hardware (semiconductor) data storage device that may be programmed similar to read/write memory but that cannot be erased without destroying the device; the stored data may be read, but not changed.

READ/WRITE MEMORY: A hardware (semiconductor) data storage device in which the stored data may be read as well as changed; common usage refers to R/W memories as random-access memories (RAMs).

REAL-TIME: The performance of a computation during the actual time that the related physical process transpires.

REGISTER: A hardware device used to store a computer word, where the word is to be manipulated as either data or an instruction.

ROUTINE: A set of coded instructions arranged in proper sequence to direct the processor to perform a desired operation or series of operations.

SIGN BIT: The bit position in a computer used to designate the algebraic sign of the word.

SHIFT: To move an ordered set of bits one or more places to the right or left.

SOFTWARE: The totality of programs and routines used to extend the capabilities of computers (such as compilers, assemblers, routines, and subroutines).

SOURCE LANGUAGE: The high-level (often mnemonic) language in which you specify a program for the computer. It is translated (by Assembler or Compiler programs) to a machine-readable binary code.

STORAGE: Any device into which units of information can be copied.

SUBROUTINE: A series of computer instructions that performs a specific task for many other routines.

WORD: An ordered set of characters that occupies a single storage location and is treated by the computer circuits as a unit and transferred as such.

WRITE: To transfer information to a device.

APPENDIX B – POSITIVE POWERS OF TWO

n	2^n				n	2^n						
1	2				51	22517	99813	68524	8			
2	4				52	45035	99627	37049	6			
3	8				53	90071	99254	74099	2			
4	16				54	18014	39850	94819	84			
5	32				55	36028	79701	89639	68			
6	64				56	72057	59403	79279	36			
7	128				57	14411	51880	75855	872			
8	256				58	28823	03761	51711	744			
9	512				59	57646	07523	03423	488			
10	1024				60	11529	21504	60684	6976			
11	2048				61	23058	43009	21369	3952			
12	4096				62	46116	86018	42738	7904			
13	8192				63	92233	72036	85477	5808			
14	16384				64	18446	74407	37095	51616			
15	32768				65	36893	48814	74191	03232			
16	65536				66	73786	97629	48382	06464			
17	13107	2			67	14757	39525	89676	41292	8		
18	26214	4			68	29514	79051	79352	82585	6		
19	52428	8			69	59029	58103	58705	65171	2		
20	10485	76			70	11805	91620	71741	13034	24		
21	20971	52			71	23611	83241	43482	26068	48		
22	41943	04			72	47223	66482	86964	52136	96		
23	83886	08			73	94447	32965	73929	04273	92		
24	16777	216			74	18889	46593	14785	80854	784		
25	33554	432			75	37778	93186	29571	61709	568		
26	67108	864			76	75557	86372	59143	23419	136		
27	13421	7728			77	15111	57274	51828	64683	8272		
28	26843	5456			78	30223	14549	03657	29367	6544		
29	53687	0912			79	60446	29098	07314	58735	3088		
30	10737	41824			80	12089	25819	61462	91747	06176		
31	21474	83648			81	24178	51639	22925	83494	12352		
32	42949	67296			82	48357	03278	45851	66988	24704		
33	85899	34592			83	96714	06556	91703	33976	49408		
34	17179	86918	4		84	19342	81311	38340	66795	29881	6	
35	34359	73836	8		85	38685	62622	76681	33590	59763	2	
36	68719	47673	6		86	77371	25245	53362	67181	19526	4	
37	13743	89534	72		87	15474	25049	10672	53436	23905	28	
38	27487	79069	44		88	30948	50098	21345	06872	47810	56	
39	54975	58138	88		89	61897	00196	42690	13744	95621	12	
40	10995	11627	776		90	12379	40039	28538	02748	99124	224	
41	21990	23255	552		91	24758	80078	57076	05497	98248	448	
42	43980	46511	104		92	49517	60157	14152	10995	96496	896	
43	87960	93022	208		93	99035	20314	28304	21991	92993	792	
44	17592	18604	4416		94	19807	04062	85660	84398	38598	7584	
45	35184	37208	8832		95	39614	08125	71321	68796	77197	5168	
46	70368	74417	7664		96	79228	16251	42643	37593	54395	0336	
47	14073	74883	55328		97	15845	63250	28528	67518	70879	00672	
48	28147	49767	10656		98	31691	26500	57057	35037	41758	01344	
49	56294	99534	21312		99	63382	53001	14114	70074	83516	02688	
50	11258	99906	84262	4	100	12676	50600	22822	94014	96703	20537	6
					101	25353	01200	45645	88029	93406	41075	2

APPENDIX C – NEGATIVE POWERS OF TWO

n	2^{-n}									
0	1.0									
1	0.5									
2	0.25									
3	0.125									
4	0.0625									
5	0.03125									
6	0.01562	5								
7	0.00781	25								
8	0.00390	625								
9	0.00195	3125								
10	0.00097	65625								
11	0.00048	82812	5							
12	0.00024	41406	25							
13	0.00012	20703	125							
14	0.00006	10351	5625							
15	0.00003	05175	78125							
16	0.00001	52587	89062	5						
17	0.00000	76293	94531	25						
18	0.00000	38146	97265	625						
19	0.00000	19073	48632	8125						
20	0.00000	09536	74316	40625						
21	0.00000	04768	37158	20312	5					
22	0.00000	02384	18579	10156	25					
23	0.00000	01192	09289	55078	125					
24	0.00000	00596	04644	77539	0625					
25	0.00000	00298	02322	38769	53125					
26	0.00000	00149	01161	19384	76562	5				
27	0.00000	00074	50580	59692	38281	25				
28	0.00000	00037	25290	29846	19140	625				
29	0.00000	00018	62645	14923	09570	3125				
30	0.00000	00009	31322	57461	54785	15625				
31	0.00000	00004	65661	28730	77392	57812	5			
32	0.00000	00002	32830	64365	38696	28906	25			
33	0.00000	00001	16415	32182	69348	14453	125			
34	0.00000	00000	58207	66091	34674	07226	5625			
35	0.00000	00000	29103	83045	67337	03613	28125			
36	0.00000	00000	14551	91522	83668	51806	64062	5		
37	0.00000	00000	07275	95761	41834	25903	32031	25		
38	0.00000	00000	03637	97880	70917	12951	66015	625		
39	0.00000	00000	01818	98940	35458	56475	83007	8125		
40	0.00000	00000	00909	49470	17729	28237	91503	90625		
41	0.00000	00000	00454	74735	08864	64118	95751	95312	5	
42	0.00000	00000	00227	37367	54432	32059	47875	97656	25	
43	0.00000	00000	00113	68683	77216	16029	73937	98828	125	
44	0.00000	00000	00056	84341	88608	08014	86968	99414	0625	
45	0.00000	00000	00028	43170	94304	04007	43484	49707	03125	
46	0.00000	00000	00014	21085	47152	02003	71742	24853	51562	5
47	0.00000	00000	00007	10542	73576	01001	85871	12426	75781	25
48	0.00000	00000	00003	55271	36788	00500	92935	56213	37890	625
49	0.00000	00000	00001	77635	68394	00250	46467	78106	68945	3125
50	0.00000	00000	00000	88817	84197	00125	23233	89053	34472	65625

APPENDIX D – THE HEXADECIMAL NUMBER SYSTEM

We have been taught from childhood to recognize and manipulate a number system called decimal or base-10, which uses ten symbols to represent values or numbers. These symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Combinations of these form other numbers, and each number or digit position is assigned a value equal to .its position in the number sequence. For example, the number 12,045:

POSITION NO.	4 3 2 1 0		
	1 2 0 4 5		
		= 5×10^0 =	5
		= 4×10^1 =	40
		= 0×10^2 =	000
		= 2×10^3 =	2,000
		= 1×10^4 =	10,000
			$12{,}045_{10}$

10 is the base-value of the number system, and 0, 1, 2, 3, 4 are the positions of weighted values.

Most computers use a base-2 numbering system in which zeros and ones are the only symbols used to represent any number. The least-significant bit would have a value of 2^0, the next bit would be 2^1, then 2^2, etc. Let's use a group of five bits and assign bit 0 as the least significant bit.

BIT NO.						
0		1		1×2^0		1
1		0		0×2^1		0
2	=	1	=	1×2^2	=	4
3		0		0×2^3		0
4		1		1×2^4		16
						21_{10}

21 is the sum of the values of the bit positions.

It can also be seen that by using larger groups of bits, larger numbers may be represented. An eight-bit computer, which can handle eight bit positions in parallel, can represent numbers from 0 to 255_{10}.

All Bits Equal 0

BIT NO.						
0		0		0×2^0		0
1		0		0×2^1		0
2		0		0×2^2		0
3	=	0	=	0×2^3	=	0
4		0		0×2^4		0
5		0		0×2^5		0
6		0		0×2^6		0
7		0		0×2^7		0
						0_{10}

All Bits Equal 1

BIT NO.						
0		1		1×2^0		1
1		1		1×2^1		2
2		1		1×2^2		4
3	=	1	=	1×2^3	=	8
4		1		1×2^4		16
5		1		1×2^5		32
6		1		1×2^6		64
7		1		1×2^7		128
						255_{10}

A computer that has 16 bit positions may represent numbers with values from zero to 65,535.

Another consideration in computers is the representation of both positive and negative values. In the "sign magnitude" system, this may be accomplished by assigning one of the bits in a group as a plus/minus indicator. The normal method is to assign the most-significant bit position to this task. If it is a logic zero, then the value is positive; if it is a logic one, then the value is minus. Assuming a group of eight bits maximum, and using the eighth position as the sign, we may represent the following numbers:

BIT NO.						
0		1		1×2^0		1
1		1		1×2^1		2
2		1		1×2^2		4
3	=	1	=	1×2^3	=	8
4		1		1×2^4		16
5		1		1×2^5		32
6		1		1×2^6		64
sign bit 7		0	=	+		$+127_{10}$

If bit 7 is equal to a 1, then the above number would be negative: –127. Note that by using the most-significant bit for the sign, the maximum number that may be represented is only ±127. In a 16-bit computer this number would be ±32,767.

Because it is difficult for us to convert visually many ones and zeros to their represented value, other methods of representing numbers have been implemented.

BCD OR BINARY CODED DECIMAL:

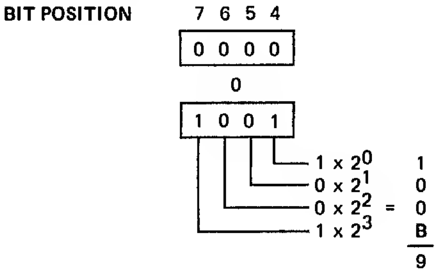
BCD uses groups of four binary bits or positions, and only uses those combinations that add up to 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. For example:

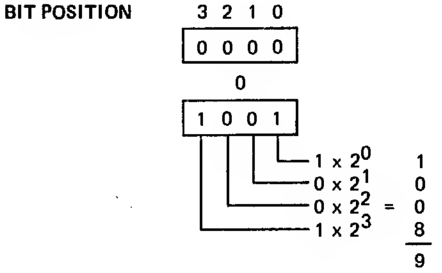
BIT	3	2	1	0		
	0	0	0	0	=	0
	0	0	0	1	=	1
	0	0	1	0	=	2
	0	0	1	1	=	3
	0	1	0	0	=	4
	0	1	0	1	=	5
	0	1	1	0	=	6
	0	1	1	1	=	7
	1	0	0	0	=	8
	1	0	0	1	=	9

The other binary combinations possible in the four bit positions are not allowed in the BCD method:

1	0	1	0	Not Valid
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

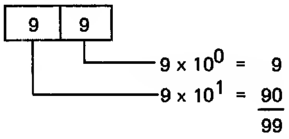
In an 8-bit computer, the decimal numbers 00 through 99 may be represented:





Note that the binary weighting system repeats for each four-bit group.

This is then compensated for by applying the decimal (base-10) rules to the converted numbers:



(By having to weigh only up to four binary bits, you quickly become efficient at converting binary numbers to decimal form and decimal numbers to binary form.)

The maximum numbers that can be represented in an 8-bit machine is then only 99_{10} in decimal versus 255_{10} in binary.

As you can see, the efficiency of a computer is restricted because of the illegal combination in each four-bit group. Another representation of binary numbers allows for *all* combinations of the four-bit groups. This system is called hexadecimal representation.

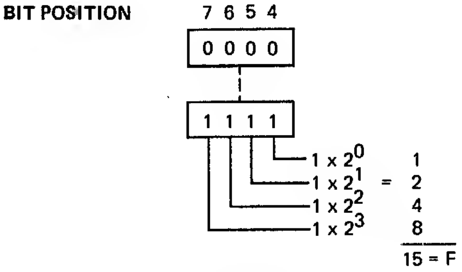
HEXADECIMAL (HEX) NOTATION

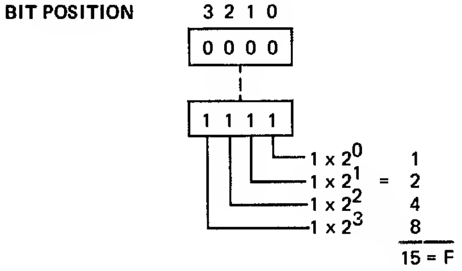
Hex uses a numbering system of base 16, and allows for all combinations of the four-bit binary groups, as follows:

BIT POSITION: 3 2 1 0	BINARY	HEX SYMBOL
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	10	A
1 0 1 1	11	B
1 1 0 0	12	C
1 1 0 1	13,	D
1 1 1 0	14	E
1 1 1 1	15	F

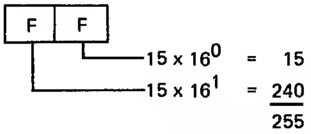
The notations A through F are used to allow for a single-character representation of the four-bit group without duplication.

With hex we can now represent all 16 combinations of binary weights possible in a group of four bit positions. An eight bit computer can then represent the numbers 00 through FF, which is equivalent to binary 0 through 255:





Applying the same rules as for decimal, but using the base 16 instead of base 10:



Thus, binary numbers, no matter what the number of position, can easily be converted simply by dividing them up into groups of four bits. For example, in a 16-bit computer:

Hex	F	E	9	A
	Λ	Λ	Λ	Λ
Binary	1111	1110	1001	1010
	V	V	V	V
Hex	F	E	9	A

Further, the use of hex symbols as an equivalent for four binary bits requires fewer printed symbols, and most computer documentation today uses the hexadecimal code representation.

POSITIVE AND NEGATIVE NUMBERS:

In hex or in binary, the method of representing positive and negative numbers is the same. The most-significant bit of the most-significant group is set to a zero for a positive number or a one for a negative number.

If there are four groups of 4-bits each, as in a 16-bit computer, we could have:

Hex	7	F	F	F
	Λ	Λ	Λ	Λ
Binary	0111	1111	1111	1111

sign bit

This number is equivalent to +32,767.

By making the most-significant-bit a logic 1, then the number becomes:

```
 F      F      F      F
 ^      ^      ^      ^
1111   1111   1111   1111
↑
└── sign bit
```

This number is equivalent to −32,767.

The method used to represent a negative hexadecimal number depends on the type of numbering system chosen for binary arithmetic processing. Most digital computers use either the "sign magnitude" system or the twos-complement system. In the sign magnitude system, a negative value is formed by setting a sign bit—the most-significant bit of the most-significant group of bits—to one, and the remaining bits to the desired absolute value. Thus, −32,767 is represented as 1111 1111 1111 1111.

Conversely, if the most-significant-bit is a zero the number is positive; +32,767 is represented as 0111 1111 1111 1111.

In the twos-complement system—the system used in PACE—positive numbers are represented exactly as in the sign magnitude system (sign bit is a logic zero); but negative numbers are represented by the twos-complement of the absolute value of the number. Thus, −32,767 becomes, in the twos-complement system, 1000 0000 0000 0001. Appendix E shows how this conversion is accomplished.

APPENDIX E – NEGATIVE HEXADECIMAL NUMBERS

The PACE microprocessor maintains negative numbers in twos-complement form. To convert a number in hexadecimal notation to its twos-complement equivalent, subtract the number from hexadecimal 2^n, where "n" is the number of binary bits in the computer word. For a 16-bit word, "n" is 16, and 2^n is 1 0000 0000 0000 0000 (binary) or 1 0000 (hex).

Thus, the negative of 1245_{16} is:

$$\begin{array}{r} 1\,0\,0\,0\,0 \\ -1\,2\,4\,5 \\ \hline E\,D\,B\,B \end{array}$$

A hexadecimal number will be negative in the PACE CPU if the left-most digit is 8, 9, A, B, C, D, E, or F (because all of these groupings start with a one). Thus, the twos-complement of hex FACE is:

$$\begin{array}{r} 1\,0\,0\,0\,0 \\ -F\,A\,C\,E \\ \hline +0\,5\,3\,2 \end{array}$$

Perhaps an easier way to find the twos-complement of a hexadecimal number is first to take the ones-complement of the number; the ones-complement plus one is the twos-complement. The ones-complement of a number is its inverted form; simply exchange its ones for zeros, and its zeros for ones. Thus,

hexadecimal		binary equivalent		ones-complement
FACE	→	1111 1010 1100 1110	→	0000 0101 0011 0001

ones-complement +1

$$\begin{array}{r} 0000\ \ 0101\ \ 0011\ \ 0001 \\ +1 \\ \hline \underbrace{0000}\ \ \underbrace{0101}\ \ \underbrace{0011}\ \ \underbrace{0010} \end{array}$$

Hex twos-complement of FACE → 0 5 3 2

APPENDIX F – HEXADECIMAL AND DECIMAL INTEGER CONVERSION TABLE

8		7		6		5		4		3		2		1	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268 435 456	1	16 777 218	1	1 048 576	1	65 536	1	4 096	1	256	1	16	1	1
2	536 870 912	2	33 554 432	2	2 097 152	2	131 072	2	8 192	2	512	2	32	2	2
3	805 306 368	3	50 331 648	3	3 145 728	3	196 608	3	12 288	3	768	3	48	3	3
4	1 073 741 824	4	67 108 864	4	4 194 304	4	262 144	4	16 384	4	1 024	4	64	4	4
5	1 342 177 280	5	83 888 080	5	5 242 880	5	327 680	5	20 480	5	1 280	5	80	5	5
8	1 610 612 736	6	100 663 296	6	6 291 456	6	393 216	6	24 576	6	1 536	6	96	6	6
7	1 879 048 192	7	117 440 512	7	7 340 032	7	458 752	7	28 872	7	1 792	7	112	7	7
8	2 147 483 648	8	134 217 728	8	8 388 608	8	624 288	8	32 768	8	2 048	8	128	8	8
9	2 415 919 104	9	150 994 944	9	9 437 184	9	589 824	9	36 864	9	2 304	9	144	9	9
A	2 684 354 560	A	167 772 160	A	10 485 760	A	655 360	A	40 960	A	2 560	A	160	A	10
8	2 952 790 016	8	184 549 376	8	11 534 336	8	720 896	8	45 056	8	2 816	8	176	8	11
C	3 221 225 472	C	201 326 592	C	12 582 912	C	786 432	C	49 152	C	3 072	C	192	C	12
D	3 489 660 928	D	218 103 808	D	13 831 488	D	851 968	D	63 248	D	3 328	D	208	D	13
E	3 758 096 384	E	234 881 024	E	14 680 064	E	917 504	E	67 344	E	3 584	E	224	E	14
F	4 028 531 840	F	251 658 240	F	15 728 640	F	983 040	F	61 440	F	3 840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

```
D =    13
      x16
      ---
      208
3 =    +3
      ---
      211
      x16
     ----
     3376
4 =    +4
     ----
     3380
```

EXAMPLE

Conversion of Hexadecimal Value	D34
D	3328
3	48
4	4
Decimal	3380

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = D34_{16}$

```
16 |3380          remainder
              4       ↑
16 |211               |
              3       |
16 |13                |
              D       |
```

EXAMPLE

Conversion of Decimal Value	3380
D	–3328
	52
3	–48
	4
4	–4
Hexadecimal	D34

APPENDIX G – HEXADECIMAL AND DECIMAL FRACTION CONVERSION TABLE

1		2		3		4	
HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL EQUIVALENT
.0	.0000	.00	.0000 0000	.000	.0000 0000 0000	.0000	.0000 0000 0000 0000
.1	.0625	.01	.0039 0625	.001	.0002 4414 0625	.0001	.0000 1525 8789 0625
.2	.1250	.02	.0078 1250	.002	.0004 8828 1250	.0002	.0000 3051 7578 1250
.3	.1875	.03	.0117 1875	.003	.0007 3242 1875	.0003	.0000 4577 6367 1875
.4	.2500	.04	.0156 2500	.004	.0009 7656 2500	.0004	.0000 6103 5156 2500
.5	.3125	.05	.0195 3125	.005	.0012 2070 3125	.0005	.0000 7629 3945 3125
.6	.3750	.06	.0234 3750	.006	.0014 6484 3750	.0006	.0000 9155 2734 3750
.7	.4375	.07	.0273 4375	.007	.0017 0898 4375	.0007	.0001 0681 1523 4375
.8	.5000	.08	.0312 5000	.008	.0019 5312 5000	.0008	.0001 2207 0312 5000
.9	.5625	.09	.0351 5625	.009	.0021 9726 5625	.0009	.0001 3732 9101 5625
.A	.6250	.0A	.0390 6250	.00A	.0024 4140 6250	.000A	.0001 5258 7890 6250
.B	.6875	.0B	.0429 6875	.00B	.0026 8554 6875	.000B	.0001 6784 6679 6875
.C	.7500	.0C	.0468 7500	.00C	.0029 2968 7500	.000C	.0001 8310 5468 7500
.D	.8125	.0D	.0507 8125	.00D	.0031 7382 8125	.000D	.0001 9836 4257 8125
.E	.8750	.0E	.0546 8750	.00E	.0034 1796 8750	.000E	.0002 1362 3046 8750
.F	.9375	.0F	.0585 9375	.00F	.0036 6210 9375	.000F	.0002 2888 1835 9375
1		2		3		4	

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A	in position 1	.6250
Find .0B	in position 2	.0429 6875
Find .00C	in position 3	.0029 2968 7500
.ABC Hex is equal to		.6708 9843 7500

APPENDIX H – INTEGER CONVERSION TABLE

POWERS OF 16

Example: $268{,}435{,}456_{10} = (2.68435456 \times 10^8)_{10} = 1000\ 0000_{16} = (10^7)_{16}$

16^n							n
						1	0
						16	1
						256	2
					4	096	3
					65	536	4
				1	048	576	5
				16	777	216	6
				268	435	456	7
			4	294	967	296	8
			68	719	476	736	9
		1	099	511	627	776	10 = A
		17	592	186	044	416	11 = B
		281	474	976	710	656	12 = C
	4	503	599	627	370	496	13 = D
	72	057	594	037	927	936	14 = E
1	152	921	504	606	846	976	15 = F

Decimal Values

APPENDIX I – OP CODE INDEX OF INSTRUCTIONS

ALPHANUMERIC SEQUENCE BY HEXADECIMAL

Read down then right.

Mnemonic Assembler Code																			
		AC0	AC1	AC2	AC3	BASE PAGE (XX)	PC REL (XX+PC)	AC2 REL (XX+AC2)	AC3 REL (XX+AC3)										
HALT	0000																	Halt	
CFR r		0400	0500	0600	0700													Copy flags to register	
CRF r		0800	0900	0A00	0B00													Copy register to flags	
PUSHF	0C00																	Push flags onto stack	
PULLF	1000																	Pull stack into flags	
JSR disp(xr)						14XX	15XX	16XX	17XX									Jump to subroutine; XX = ±127; push PC onto stack	
JMP disp(xr)						18XX	19XX	1AXX	1BXX									Jump; XX = ±127	
XCHRS r		1C00	1D00	1E00	1F00													Exchange register and stack	
RDL r,n,l		20XX	21XX	22XX	23XX													Rotate register left	
RDR r,n,l		24XX	25XX	26XX	27XX													Rotate register right	Bit 1 = 1 include link bit
SHL r,n,l		28XX	29XX	2AXX	2BXX													Shift left	Bit 2 = 2 shift count
SHR r,n,l		2CXX	2DXX	2EXX	2FXX													Shift right	Bits 2-7 = N = shift count
fc		NDT USED	IE1	IE2	IE3	IE4	IE5	DVF	CRY	LINK	IEN	BYTE	F11	F12	F13	F14	NDT USED		
PFLG fc		3000	3100	3200	3300	3400	3500	3600	3700	3800	3900	3A00	3B00	3C00	3D00	3E00	3F00	Pulse or reset flag	
SFLG fc		3080	3180	3280	3380	3480	3580	3680	3780	3880	3980	3A80	3B80	3C80	3D80	3E80	3F80	Set flag	
cc		STACK Full	AC0 = 0	AC0 Bit15=0	AC0 Bit0=1	AC0 8it1=1	AC0 ≠0	AC0 Bit2=1	CDNT	LINK	IEN	CRY	AC0 8it 15=0	DVF	JC13	JC14	JC15		
BDC cc,disp		40XX	41XX	42XX	43XX	44XX	45XX	46XX	47XX	48XX	49XX	4AXX	48XX	4CXX	4DXX	4EXX	4FXX	Branch on condition (PC relative) XX = ±127	
		AC0	AC1	AC2	AC3														
LI r, disp		50XX	51XX	52XX	53XX													Load immediate; load register with XX; XX = data Bit 7 of XX extends to Bits 8-15 of register	
sr		AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3		
dr		AC0	AC0	AC0	AC0	AC1	AC1	AC1	AC1	AC2	AC2	AC2	AC2	AC3	AC3	AC3	AC3		
RAND sr,dr		5400	5440	5480	54C0	5500	5540	5580	55C0	5600	5640	5680	56C0	5700	5740	5780	57C0	"AND" register to register; result to register (dr)	
RXDR sr,dr		5800	5840	5880	58C0	5900	5940	5980	59C0	5A00	5A40	5A80	5AC0	5B00	5B40	5B80	58C0	Exclusive "D R" register to register; result to register (dr)	
RCPY sr,dr		5C00	5C40	5C80	5CC0	5D00	5D40	5D80	5DC0	5E00	5E40	5E80	5EC0	5F00	5F40	5F80	5FC0	Copy register to register	
		AC0	AC1	AC2	AC3														
PUSH r		6000	6100	6200	6300													Push register onto stack	
PULL r		6400	6500	6600	6700													Pull stack into stack	
sr		AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3		
dr		AC0	AC0	AC0	AC0	AC1	AC1	AC1	AC1	AC2	AC2	AC2	AC2	AC3	AC3	AC3	AC3		
RADD sr,dr		6800	6840	6880	68C0	6900	6940	6980	69C0	6A00	6A40	6A80	6AC0	6B00	6B40	6B80	6BC0	Add register to register; result to register (dr), overflow, and carry	
RXCH sr,dr		6C00	6C40	6C80	6CC0	6D00	6D40	6D80	6DC0	6E00	6E40	6E80	6EC0	6F00	6F40	6F80	6FC0	Exchange register	
		AC0	AC1	AC2	AC3														
CAI r, disp		70XX	71XX	72XX	73XX													Complement register and add XX; result to register Bit 7 of XX is extended to Bits 8-15	
sr		AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3	AC0	AC1	AC2	AC3		
dr		AC0	AC0	AC0	AC0	AC1	AC1	AC1	AC1	AC2	AC2	AC2	AC2	AC3	AC3	AC3	AC3		
RADC sr,dr		7400	7440	7480	74C0	7500	7540	7580	75C0	7600	7640	7680	76C0	7700	7740	7780	77C0	Add register to register plus carry; result to register (dr); overflow and carry	

APPENDIX I (Continued) OP CODE INDEX OF INSTRUCTIONS

ALPHANUMERIC SEQUENCE BY HEXADECIMAL

Read down then right.

Mnemonic Assembler Code		AC0	AC1	AC2	AC3	BASE PAGE XX	PC REL (XX+PC)	AC2 REL (XX+AC2)	AC3 REL (XX+AC3)	
AISZ r, disp		78XX	79XX	7AXX	7BXX					Add XX to register; skip next instruction if result = zero; XX = ±127
RTI disp	7CXX									Return from interrupt; add XX to top of stack and place result in PC; XX = ±127; set IEN flag
RTS disp	80XX									Return from subroutine; add XX to top of stack and place result in PC; XX = ±127
DECA 0, disp(xr)						88XX	89XX	8AXX	8BXX	Decimal add register AC0 to contents of effective address; result to AC0, overflow and carry; address = (XX + register shown); XX = ±127
ISZ disp(xr)						8CXX	8DXX	8EXX	8FXX	Increment contents of effective address by 1; skip next instruction if result = 0; result is in EA; use address mode shown; XX = ±127
SU88 0, disp(xr)						90XX	91XX	92XX	93XX	Subtract contents of effective address from AC0; result to AC0; use address mode shown; XX = ±127
JSR @ disp(xr)						94XX	95XX	96XX	97XX	Jump to subroutine indirect; push PC onto stack; final address = to contents of location (XX + register shown); XX = ±127
JMP @ disp(xr)						98XX	99XX	9AXX	9BXX	Jump indirect; final address = to contents of location (XX + register shown); XX = ±127
SKG 0, disp(xr)						9CXX	9DXX	9EXX	9FXX	Compare AC0 with contents of location (XX + register shown); XX = ±127; skip next instruction if AC0 > (EA)
LD 0, @ disp(xr)						A0XX	A1XX	A2XX	A3XX	Load indirect; load AC0 with contents of final address; address = contents of location (XX + register shown); XX = ±127
DR 0, disp(xr)						A4XX	A5XX	A6XX	A7XX	DR AC0 with contents of location (XX + register shown); XX = ±127; result to AC0
AND 0, disp(xr)						A8XX	A9XX	AAXX	ABXX	AND AC0 with contents of location (XX + register shown); XX = ±127; result to AC0
DSZ disp(xr)						ACXX	ADXX	AEXX	AFXX	Decrement contents of effective address by 1; skip next instruction if result = 0; result is in EA; address = (XX + register shown); XX = ±127
ST 0, @ disp(xr)						B0XX	B1XX	B2XX	B3XX	Store indirect; store AC0 into final address; address = cootents of location (XX + register shown); XX = ±127
SKAZ 0, disp(xr)						B8XX	B9XX	BAXX	BBXX	AND AC0 with contents of location (XX + register shown); skip next instruction if result = 0; XX = ±127
LSEX 0, disp(xr)						BCXX	BDXX	BEXX	BFXX	Load AC0 with sign extended; Bit 7 of location (XX + register shown) is extended to AC0 8-15; Bits 0-7 are loaded to AC0 Bits 0-7; XX = ±127
LD r, disp(xr)		╳				C0XX	C1XX	C2XX	C3XX	Load AC0 with contents of location (XX + register shown); XX = ±127
			╳			C4XX	C5XX	C6XX	C7XX	Load AC1 with contents of location (XX + register shown); XX = ±127
				╳		C8XX	C9XX	CAXX	CBXX	Load AC2 with contents of location (XX + register shown); XX = ±127
					╳	CCXX	CDXX	CEXX	CFXX	Load AC3 with contents of location (XX + register shown); XX = ±127
ST r, disp(xr)		╳				D0XX	D1XX	D2XX	D3XX	Store AC0 to location (XX + register shown); XX = ±127
			╳			D4XX	D5XX	D6XX	D7XX	Store AC1 to location (XX + register shown); XX = ±127
				╳		D8XX	D9XX	DAXX	DBXX	Store AC2 to location (XX + register shown); XX = ±127
					╳	DCXX	DDXX	DEXX	DFXX	Store AC3 to location (XX + register shown); XX = ±127
ADD r, disp(xr)		╳				E0XX	E1XX	E2XX	E3XX	Add AC0 to location (XX + register shown); XX = ±127; result to AC0
			╳			E4XX	E5XX	E6XX	E7XX	Add AC1 to location (XX + register shown); XX = ±127; result to AC1
				╳		E8XX	E9XX	EAXX	EBXX	Add AC2 to location (XX + register shown); XX = ±127; result to AC2
					╳	ECXX	EDXX	EEXX	EFXX	Add AC3 to location (XX + register shown); XX = ±127; result to AC3
SKNE r, disp(xr)		╳				F0XX	F1XX	F2XX	F3XX	Compare AC0 to location (XX + register shown); XX = ±127; if not equal skip next instruction
			╳			F4XX	F5XX	F6XX	F7XX	Compare AC1 to location (XX + register shown); XX = ±127; if not equal skip next instruction
				╳		F8XX	F9XX	FAXX	FBXX	Compare AC2 to location (XX + register shown); XX = ±127; if not equal skip next instruction
					╳	FCXX	FDXX	FEXX	FFXX	Compare AC3 to location (XX + register shown); XX = ±127; if not equal skip next instruction